A TRIDENT SCHOLAR PROJECT REPORT

NO. 473

Creating a Fast SMT Solver for the Theory of Real Non-Linear Constraints

by

Midshipman 1/C Fernando R. Vale-Enriquez, USN

UNITED STATES NAVAL ACADEMY
ANNAPOLIS, MARYLAND

USNA-1531-2

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
05-21-18		

4. TITLE AND SUBTITLE
Creating a Fast SMT Solver for the Theory of Real Non-Linear Constraints

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Vale-Enriquez, Fernando R.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Naval Academy
Annapolis, MD 21402

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Trident Scholar Report no. 473 (2018)

12. DISTRIBUTION / AVAILABILITY STATEMENT

This document has been approved for public release; its distribution is UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The satisfiability problem asks whether some given logical formula, for any valid combination of input, can ever evaluate to true. Satisfiability Modulo Theory (SMT) Solvers are specialized software tools which answer the satisfiability problem for a formula in some theory, such as the theory of real numbers. SMT Solvers are the core of many important tools and fields, such as automated theorem proving, hybrid systems design, formal verification of software, and security design tools. Tools which use SMT Solvers have been shown to be highly effective, but the difficulty of satisfiability solving in certain relevant theories limits the usefulness of SMT Solvers in industry. SMT Solvers use a model which incorporates two different pieces of software, a satisfiability solver and a theory solver. A solver is used to do solving on the logical structure of a problem, and generates a list of logical assumptions which satisfies the structure of the problem. The SMT solver translates these assumptions into facts in the theory, which is checked for satisfiability by the theory solver. The efficiency of theory solvers is typically the limiting factor in the speed of SMT Solvers. One poorly phrased query can take a theory solver hours to solve. To improve the efficiency of SMT Solvers, we propose a new model of SMT Solver which incorporates a partial theory solver. The new partial theory solver solves many, but not all problems very quickly. In fact, it is guaranteed to run in polynomial time in all cases. This limits the time which a slower yet fully-edged must be applied; the partial theory solver solves in fractions of a second what may take a full theory solver a signficant amount of time. We developed our theory solver by augmenting algorithms discussed in the paper Black-box/white-box simplification and applications to quantifier elimination by Christopher W. Brown and Adam W. Strzebonski. We modified these algorithms so that they could be used in the context of SMT Solving. This process required modifying the algorithms so that they output a reason for every deduction they made. Finally, we implemented these algorithms into a theory solver, and developed a full SMT solver which incorporates the partial theory solver.

15. SUBJECT TERMS
satisfiability, SMT, first-order logic

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**		54	**19b. TELEPHONE NUMBER** *(include area code)*

U.S.N.A. --- Trident Scholar project report; no. 473 (2018)

Creating a Fast SMT Solver for the Theory of Real Non-Linear Constraints

by

Midshipman 1/C Fernando R. Vale-Enriquez
United States Naval Academy
Annapolis, Maryland

_____
(signature)

Certification of Adviser(s) Approval

Professor Christopher W. Brown
Computer Science Department

_____
(signature)

_____
(date)

Acceptance for the Trident Scholar Committee

Professor Maria J. Schroeder
Associate Director of Midshipman Research

_____
(signature)

_____
(date)

Abstract

The satisfiability problem asks whether some given logical formula, for any valid combination of input, can ever evaluate to true. Satisfiability Modulo Theory (SMT) Solvers are specialized software tools which answer the satisfiability problem for a formula in some theory, such as the theory of real numbers. SMT Solvers are the core of many important tools and fields, such as automated theorem proving, hybrid systems design, formal verification of software, and security design tools. Tools which use SMT Solvers have been shown to be highly effective, but the difficulty of satisfiability solving in certain relevant theories limits the usefulness of SMT Solvers in industry.

SMT Solvers use a model which incorporates two different pieces of software, a satisfiability solver and a theory solver. A solver is used to do solving on the logical structure of a problem, and generates a list of logical assumptions which satisfies the structure of the problem. The SMT solver translates these assumptions into facts in the theory, which is checked for satisfiability by the theory solver. The efficiency of theory solvers is typically the limiting factor in the speed of SMT Solvers. One poorly phrased query can take a theory solver hours to solve.

To improve the efficiency of SMT Solvers, we propose a new model of SMT Solver which incorporates a partial theory solver. The new partial theory solver solves many, but not all problems very quickly. In fact, it is guaranteed to run in polynomial time in all cases. This limits the time which a slower yet fully-fledged must be applied; the partial theory solver solves in fractions of a second what may take a full theory solver a significant amount of time. We developed our theory solver by augmenting algorithms discussed in the paper Black-box/white-box simplification and applications to quantifier elimination by Christopher W. Brown and Adam W. Strzebonski. We modified these algorithms so that they could be used in the context of SMT Solving. This process required modifying the algorithms so that they output a reason for every deduction they made. Finally, we implemented these algorithms into a theory solver, and developed a full SMT solver which incorporates the partial theory solver.

Keywords

satisfiability, SMT, first-order logic

Acknowledgments

This work would not have been possible without the existence of the USNA Trident Committee and all of its members. Behind the scenes is a significant administrative effort to keep the program moving forward, and for that I am thankful. Additionaly, I am very thankful for the gracious support of the Office of Naval Research for the Trident Scholar Program.

I am grateful to all of the fantastic professors and officers who have taught me at USNA. In particular, I am thankful to Major Barnes, who was my inspiration to join the USMC and to major in Computer Science, and to my Trident Adviser, Dr. Brown. Dr. Brown was the first to make me understand that Computer Science is something more than just tapping away at a computer until it does what you want, and has given me the guidance, encouragement, and infinite patience I needed to be succesful in this endeavor.

Finally, I wish to thank my friends and family for their support. I wish to thank my roommates, for keeping me humble and for tolerating my many late night coding sessions. I wish to thank my sponsors, for believing in me when things were tough. Finally, I wish to thank my father, for being my inspiration to undertake my own research, and my mother, for teaching me the self-discipline necessary for any rigorous academic project.

Contents

1 Introduction

1.1 Contributions

We have augmented the algorithms introduced in [1] and [2] to return explanations along with any deductions they make. Our augmented algorithms retain the property of the original algorithms in that they run in polynomial time. We have implemented them in a fast but incomplete theory solver, and we have tested them on strict conjunctions from the SMTLIB repository. Furthermore, we created an SMT solver that uses the fast but incomplete solver in an eager fashion in addition to a complete theory solver which we call in a lazy fashion. We tested our SMT Solver on 7434 problems from the SMTLIB repository.

1.2 Key Concepts

Throughout this paper, we make explicit references to terms such as *first-order logic*, *satisfiability*, and *propositional logic*. Since it is important to have a degree of familiarity with these concepts, we give a brief overview here.

A *First-Order theory* is a language for describing some domain of interest in a precise, logical way. Let us use take this table below as an instance of one such domain:

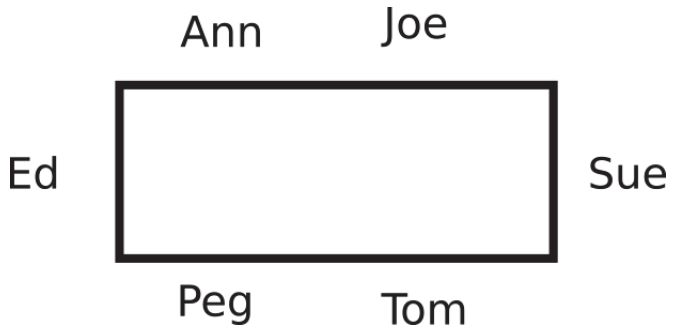

Figure 1: A table. How mundane.

The first order-theory of the table establishes rules for creating formulas that describe scenarios like this. For example, the formula below states that there are two people labeled x and y who do not sit next to each other, and that the person across from y does not sit next to x:

$$\neg IsNeighbor(x, y) \wedge \neg IsNeighbor(Across(x), y)$$

The *satisfiability* problem asks whether or not we can assign elements from the domain of interest to the variables x and y in the formula such that the formula evaluates to true. If we let $x = Ed$ and $y = Sue$, we have generated what is called a satisfying assignment. Satisfying assignments cause the formula to evaluate to true. This assignment satisfies the formula since Ed does not sit next to Sue, and the person across from Ed is Sue, and Sue cannot be her own neighbor. To *solve* in this context means to determine if a satisfying assignment exists, and if so to produce one.

For the purpose of satisfiability solving, we divide formulas like these into two components. The first is the *propositional logic* of the formula. Propositional logic is similar to first order logic, but without the components which make it about a specific domain of interest. For example, the above first-order formula in the form of propositional logic is:

$$\neg a \wedge \neg b$$

The second component is the first order logic components of the formula. This is what makes a first-order formula about a specific domain of interest. In the first order formula above, the predicate $IsNeighbor(x, y)$ and function $Across(x)$ are used to express constraints that make the formula specifically about the table. We will offer a definition of predicates and functions later in the paper, but it is important to know that these are specific to a given first-order theory.

We make the distinction between these two components because this distinction is made by the state-of-the-art tool for determining the satisfiability of formulas in a first-order theory: *Satisfiability Modulo Theory* (SMT) solvers.

1.3 SMT and the Theory of Non-Linear Constraints

SMT solvers are a framework which make use of two existing pieces of software. The first is a Propositional Satisfiability (SAT) solver, which answers the satisfiability problem for formulas in propositional logic. The second is a first order theory solver, which determines if a given set of theory constraints are simultaneously satisfiable.

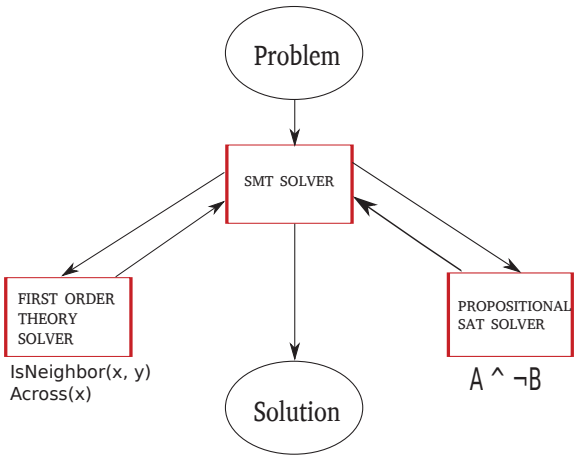

Figure 2: A model of SMT solvers

SMT solvers have gained increasing attention as of late due to algorithmic improvements that have made them fast enough to solve large scale problems [3] [4] [5]. One such application is software verification, where we make inferences about the structure of a program and determine if it will always do what we want it to do for all potential input. By modeling a program in the theory of linear constraints, SMT solvers can determine facts about a program such as invariants, or things which are always true for the entire execution of a program.

Another significant application of SMT solvers is the analysis of hybrid systems [6]. Hybrid systems include discrete modes (such as on or off) and continuous dynamic behavior. For example, a UAV might have a normal flight mode and a collision avoidance mode, where proximity to another object forces the system to enter collision avoidance mode. Hybrid system analyzers such as KeYmaera use SMT solvers as their backbone.

Other application and research areas of SMT solvers include the analysis of cryptographic models [7], mathematical theorem proving [8], and network protocol checking [9]. Many of these areas of research struggle with the same problem that we tackle in this project: the difficulty of analyzing polynomial non-linear constraints with SMT solvers. The complexity issues associated with non-linear constraints is a major obstacle for formal verification in other industries [6], and the goal of this project is to develop techniques to help allow SMT solvers to overcome this obstacle.

2 Background

In this section, we explain the components which make up SMT Solvers, and how they are used together in a working system.

2.1 Propositional logic

Propositional logic is the determination of the truth value of propositions in a statement or series of connected statements. A proposition is any statement that can be described as true or false, but not both. For example, this statement is a proposition:

MIDN Shiotani is not a firstie.

We could represent this statement as an arbitrary propositional variable, such as p. We can also combine propositions with logical operators, such as *and* ($\wedge$), *or* ($\vee$), and *not* ($\neg$). The output of the logical operators depend on the truth values of their inputs, according to their "truth tables."

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

(a) AND Table

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

(b) OR Table

p	$\neg q$
0	1
1	0

(c) NOT Table

Figure 3: Tables for AND, OR, and NOT, respectively

For example, if we let the statement *MIDN Shiotani is part of 25th company* be represented by the variable q, we can generate some example statements. MIDN Shiotani is both

a firstie and part of 25th company, so we shall let p be assigned the value false and q true. Thus, the statement $p \vee q$ evaluates to true, while $q \wedge p$ and $p \vee \neg q$ both evaluate to false.

2.2 Satisfiability Solving with the DPLL algorithm

Satisfiability (SAT) solvers are programs which read propositional formulas, and determine their "satisfiability". A propositional formula is said to be satisfiable as long as there is at least one assignment of true or false values to the propositional variables such that the formula evaluates to true. A satisyfing assignment is such an assignment. An unsatisfiable formula is a formula with no satisfying assignments.

SAT solvers typically work with propositional formulas in *Conjunctive Normal Form* (CNF). A formula in CNF is a conjunction (logical *and*) of statements called clauses, where each clause is a disjunction (logical *or*) of "literals". A literal is a propositional variable or the negation of a propositional variable. For example, the formula $(a \vee b \vee \neg c) \wedge (\neg a \vee c)$ is in CNF, while the formula $(a \wedge b \vee (c \wedge \neg d) \wedge (d \vee a))$ exhibits no clause structure and is not in CNF.

We represent propositional formulas in CNF for two reasons. The first is that any formula can be converted to CNF via the process of Tseitin transformation. Tseitin transformation is a process that takes arbitrary formulas and produces an equivalent formula with only a linear increase in size. A description of the Tseitin transformation algorithm is outside the scope of this project. It suffices to understand that because of this process, we can assume all propositional formulas we consider are in CNF without loss of generality.

In order to satisfy a CNF clause, only one literal needs the assignment of true; to satisfy the formula, all clauses must evaluate to true. This simple representation enables the use of the powerful "DPLL" algorithm for propositional SAT solving. Modern SAT solvers follow the DPLL algorithm to determine the satisfiability of formulas in CNF. The DPLL algorithm uses a stack data structure and four fundamental operations: Unit Propagation , Decide, Backtrack, and Fail.

DPLL keeps track of the assignments it has made to the propositional variables with a data structure called a stack. To understand a stack, imagine washing a large amount of plates. After a plate is finished, it is put to the side on top of the last finished plate. In computer science, we refer to this as pushing an item on the stack. When it is time to put away the plates, the upper most plate is returned first. We refer to this as popping an item off the stack.

When DPLL assigns a true or false value to a propositional variable, it pushes the assignment onto a stack data structure. DPLL assigns values with two methods. The first is by UnitPropogation (UProp), where an assignment is made based off the existing assignments and logical deductions. If all but one of the literals in a given clause have an assignment in the stack, and none of them make the clause true, then the last remaining literal must make the clause true. It is assigned false in the stack if the literal is negated, and true otherwise. The second is by decision, where some heuristic is used to assign an arbitrary value to an arbitrary variable. Decision is typically used when no unit propagation is available.

For example, consider the following formula in CNF, and the use of the stack to keep track of variable assignments. Note that $\overline{a}$ is equivalent to $\neg a$, and a^d indicates that an

assignment was generated by decision. The decision heuristic chooses random assignments for the first variable in alphabetical order:

$$\underset{C1}{(a \vee \neg b)} \wedge \underset{C2}{(a \vee \neg c)} \wedge \underset{C3}{(b \vee d)} \wedge \underset{C4}{(\neg e \vee \neg b \vee f)} \wedge \underset{C5}{(\neg a \vee \neg c \vee \neg e \vee \neg f)}$$

1. $\| \overline{a}^d$ - by decision
2. $\| \overline{a}^d \overline{b}$ - by UProp on clause 1
3. $\| \overline{a}^d \overline{b} \overline{c}$ - by UProp on clause 2
4. $\| \overline{a}^d \overline{b} \overline{c} d$ - by UProp on clause 3
5. $\| \overline{a}^d \overline{b} \overline{c} d \overline{e}^d$ - by decision
6. $\| \overline{a}^d \overline{b} \overline{c} d \overline{e}^d f^d$ - by decision *SATISFYING ASSIGNMENT*

DPLL detects what is called a "conflict" when it finds that a clause evaluates to false. The original DPLL algorithm used an operation called backtrack to resolve conflicts. Backtrack pops every assignment off of the stack until it encounters a decision variable. When it encounters the decision, it pushes the negation of that variable onto the stack but no longer marked as a decision. If backtrack pops all assignments off the stack, the algorithm fails and tells us that the propositional formula is unsatisfiable. Backtrack is a slow, inefficient process. A decision buried in the stack may be the root cause of conflicts, but backtrack must attempt to change every decision until it reaches the "buried" decision, which wastes time checking impossible assignments.

Modern solvers use an operation called "backjump," which is substantially faster than backtrack. The backjump algorithm learns the root assignments that resulted in a conflict, and adds them to the propositional formula as another clause. It uses this clause in order to clear decisions which did not contribute to the conflict from the stack, and change the assignment of one of the root causes of the conflict.

Clauses learned by backjump do not change the meaning of a propositional formula. Backjump discovers information implied by the formula, and makes it explicit so that DPLL may use it to guide its search.

Consider the previous example, where a different decision is made, resulting in a conflict:

1. $\| a^d$ - by decision
2. $\| a^d b$ - by UProp on clause 1
3. $\| a^d bc$ - by UProp on clause 2
4. $\| a^d bcd^d$ - by decision
5. $\| a^d bcd^d e^d$ - by decision
6. $\| a^d bcd^d e^d \overline{f}$ - by UProp clause 4 *CONFLICT ON CLAUSE 5*!

From this conflict, backjump learns the clause $\neg a \vee \neg e$ and adds it to the formula. In other words, the learned clause states that one of a or e must be true to acquire a satisfying assignment. Thus, the formula is now:

$$\underset{C1}{(a \vee \neg b)} \wedge \underset{C2}{(a \vee \neg c)} \wedge \underset{C3}{(b \vee d)} \wedge \underset{C4}{(\neg e \vee \neg b \vee f)} \wedge \underset{C5}{(\neg a \vee \neg c \vee \neg e \vee \neg f)} \wedge \underset{C6}{(\neg a \vee \neg e)}$$

The search resumes from state $\| \; a^d\overline{e}$ since the new clause allows us to propagate $\overline{e}$ from a^d. Notice that the decision on d is erased, as backjump enables DPLL to skip multiple decisions without wasting time exploring impossible branches:

1. $\| \; a^d\overline{e}$ - by backjump on clause 6
2. $\| \; a^d\overline{e}f$ - by UProp on clause 4
3. $\| \; a^d\overline{e}fb$ - by UProp on clause 1
4. $\| \; a^d\overline{e}fbc$ - by UProp on clause 2
5. $\| \; a^d\overline{e}fbcd^d$ - by decision *SATISFYING ASSIGNMENT!*

Backjump performs clause learning by constructing a graph, which is a data structure of interconnected nodes. A graph is similar to a road map, where one city is connected to many others by roads, and part of a greater network of interconnected cities. In this metaphor, a city is a node, and a road is an edge, the link between two cities or nodes. Backjump traces the variable assignments that lead to the current stack by examining which clause propagated each variable. In other words, each assignment needs to know which clause propagated it. Each variable acquires an edge to each variable in the clause which propagated it under the current assignment. Backtrack traces backwards for all variables on the current decision level, and uses the resulting graph to construct the learned clause via the use of specialized algorithms for graph traversal.

Computer scientists have made a great deal of other advancements in the basic DPLL algorithm. VSIDS is a heuristic which allows SAT solvers to decide on unassigned variables which frequently appear. Deciding using VSIDS increases the chance that one decision will result in many propagations. The two-watched literal scheme provides for fast and memory-efficient conflict and unit-propagation detection, as opposed to iterating through every single clause to detect a unit propagation. Random restarts prevent the DPLL algorithm from spending too much time on a decision path which will never yield a satisfying assignment. Clause forgetting allows SAT solvers to eliminate clauses which do not contribute after the associated backjump, improving memory usage and speed. The result of all these improvements and more is very fast SAT solvers which often solve huge formulas in a very short amount of time. The speed of SAT solvers is a key motivation for the DPLL(T) architecture of SMT solvers which will be introduced later.

Satisfiability solving alone outside the context of SMT solvers remains an important topic in Computer Science. For more thorough introductions to the topic, consult [10], [11], or [12].

2.3 First-Order Logic

Propositional logic is limited in the type of information it can represent. Using propositional logic, we can discuss specific objects or items, but we must have a propositional variable for every item that appears in a formula. This makes it difficult and overtly complex to represent many types of problems in propositional logic. We may generate extremely large formulas for certain types of problems, which results in inefficient SAT solving.

First-order logic, or predicate logic, is an extension of Propositional logic which allows us to describe a specific domain of interest with functions and predicates. Functions map elements within the domain of interest to other elements within the domain. Predicates evaluate properties of elements of the domain to the values of true or false. For example, if Midshipmen are the domain of interest, we could define a function called $Roommate()$ and a Predicate $IsCompanyCommander()$. Calling $Roommate$(MIDN Vale-Enriquez) will return "MIDN Shiotani." Calling $IsCompanyCommander$(MIDN Vale-Enriquez) will return false, since the author is not a company commander.

The conciseness with which we can express ideas in first-order logic as compared to propositional logic is an important factor in how precisely we map problems from the former to the latter.

We use tools called "theory solvers" to determine the satisfiability of a formula in first order logic. The implementation of each theory solver depends on the specific logic, or set of rules, that problems are represented in.

2.4 Theory of Real Non-Linear Constraints

The Theory of Real Non-Linear Constraints concerns the satisfiability of conjunctions of equalities and inequalities of real numbers. The domain of interest in this theory is the set of real numbers. The predicates in this theory are the relational operators, such as $=, \neq, \geq, >, \leq$, and $<$. The functions in this theory are mathematical operators such as $+, -, /$, and $*$. Exponential functions, a part of the algorithms presented later in this problem, can be represented with the use of the $*$ function on the same number multiple times. An example set of constraints is as follows:

$$1 - x + y^2 \neq 0 \wedge 2x - 1 < 0$$

Alfred Tarski's 1951 paper *A Decision Method for Elementary Algebra and Geometry* [13] proves that any problem like the above in the theory of real closed fields with quantifiers (i.e., $\exists$ and $\forall$) can be reduced to an equivalent problem without quantifiers. As a consequence, Tarski proved that there exists a decision procedure for proving if a formula actually is a member of the theory of real closed fields (i.e., we can answer the satisfiability problem with an algorithm). Tarski proved his theory for the theory of real closed fields, but since any problem which is expressed in the theory of real closed fields has a one to one correspondence with problems expressed in real non-linear constraints, they are one in the same.

In [14], George Collins provides an algorithm called Cylindrical Algebraic Decomposition (CAD) which allows us to do quantifier elimination and satisfiability solving for the theory of real non-linear constraints. CAD is the basis of Theory Solvers such as QEPCAD[15] and

Mathematica[16], but it has a computational complexity that is double exponential in the number of variables.

2.5 SMT Solving with DPLL(T)

Modern day SMT solvers follow the basic strategy of the "DPLL(T)" algorithm [12]. It has at least two main components, where the first is a fast SAT solver and the second is a slower theory solver. The SMT solver governs how the two components work with one another.

DPLL(T) takes advantage of the relatively faster speed of SAT solvers compared to theory solvers. The SMT framework maps the original logical expression into a propositional formula, and then uses the SAT solver to find satisfying assignments for the propositional formula. However, SAT solvers only receive approximations of the original logical formula. Consequently, a satisfying assignment proposed by the SAT solver may be found invalid by the theory solver. Additionally, the SAT solver may not recognize possible variable propagations implied by the contents of the stack for the same reason.

To better describe this, let us consider the satisfiability of the following formula in the theory of real nonlinear constraints:

$$(xy > 1 \vee x = 0) \wedge (x = 0 \vee \neg y = 0)$$

The first step an SMT solver using DPLL(T) takes is to map the formula into propositional logic. In this case, it will generate the formula:

$$(a \vee b) \wedge (b \vee \neg c)$$

Here, A stands for $xy > 1$, B stands for $x = 0$, and C stands for $y = 0$. The SMT solver then instructs the SAT solver to find a satisfying solution for this formula. It might give the solution:

$$\| \; a^d b^d$$

The SMT solver maps this formula back into theory assertions, such that the proposed solution is $xy > 1$ and $x = 0$. It then tasks the theory solver with verifying that these two assertions may be simultaneously satisfied. This is known as requesting a Theory Check, or "TCheck." In this case, it rejects the assertions and returns:

$$\neg xy > 1 \vee \neg x = 0$$

In other words, the theory solver says that these two assertions were responsible for the conflict, and one of them must be false to produce a satisfying assignment. This operation is known as a Theory Explain, or "TExplain." The SMT solver uses the learned information to construct a propositional clause of the form $(\neg A \vee \neg B)$ and adds it to the propositional formula. The propositional formula is now of the form:

$$(a \vee b) \wedge (b \vee \neg c) \wedge (\neg a \vee \neg b)$$

The propositional SAT solver uses the newly learned clause to backjump, and then discover another solution. One important feature of this step is that the new backjump clause

changes the meaning of the propositional formula. It makes it resemble the original first order formula more, such that the SAT solver is now more likely to pick solutions which are satisfiable in the first-order theory. For this reason, producing an explanation via TExplain is absolutely essential. For some formulas, a SAT solver can produce millions of satisfying assignments which do not make sense in the theory for the same reason. A single learned clause cuts off all of those assignments at once.

The SMT solver repeats this process of acquiring a propositional solution from the SAT solver and verifying it with the first order theory solver. This approach to SMT solving is known as the lazy approach. The lazy approach works best when the theory solver works slowly, such as for the theory of real non-linear constraints. If the theory solver works quickly, we design SMT solvers to use what is called the eager approach.

Eager SMT solvers constantly interrupt the Propositional SAT solver before it has acquired a propositional solution. This approach allows for four different operations:

1. It can request a Theory Check on a partial solution. Checking a partial solution allows the SMT solver to save precious time by rejecting a propositional solution before the SAT solver wastes time finishing it.

2. In the case of a conflict, it requires a TExplain such as the one stated in the previous example.

3. It can check if any variables can be added to the partial solution via Theory Propagate, or "TProp." This is very similar to the UProp operation, except it is done by the theory solver. While variables added by UProp are implied by the logical structure of a formula, variables added by TProp are implied by the rules of the first-order formula. In the previous example, if the SMT solver requested a TProp while the SAT solver only had the variable a^d on the stack, it could then add $\neg b$ and $\neg c$. This is because a represents $xy > 0$, and the truth of this assertion implies that neither x nor y equal 0. When we have very fast theory solvers, we always check if a variable may be added by TProp before we add them via UProp.

4. In the event that the SAT solver encounters a conflict, the SMT solver may request an explanation for why an assignment was added via TProp. This is known as Theory Learn, or "TLearn." This is so the SAT solver may construct a backjump clause over the normal operation of DPLL.

Eager SMT solvers are faster than lazy SMT solvers as long as the first order theory solver of the SMT solver is fast enough for the eager approach. The speed of the eager approach and the requirements it places on the theory solver motivates this Trident project.

3 Project Goals and Contributions

In this Trident project we implemented a fast theory solver for the theory of real non-linear constraints and incorporated it into an SMT Solver we designed. Existing theory solvers are complete in that they recognize all conflicts and guarantee correctness, but are too slow to

be called in a tight loop. Therefore, they must be called in a "lazy" fashion. The new theory solver we created is much faster than existing theory solvers, but is incomplete and cannot recognize all conflicts.

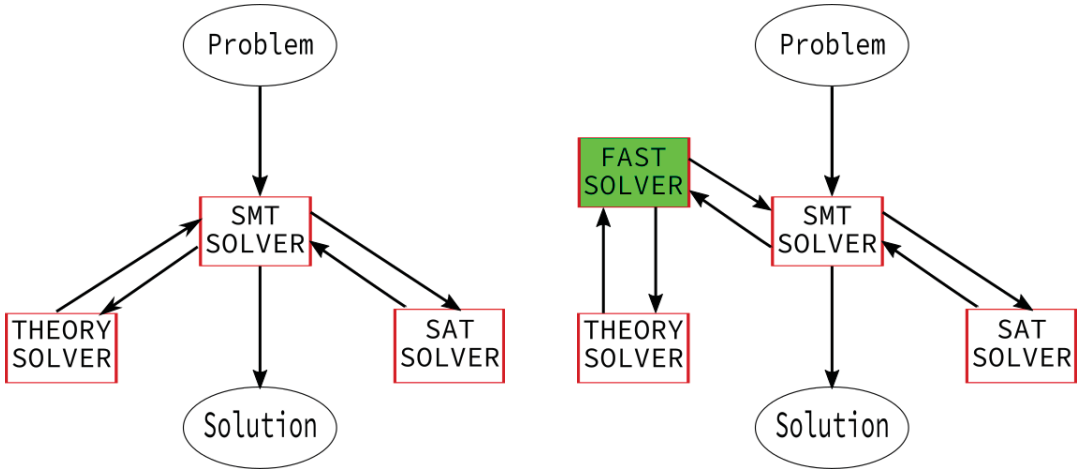

Figure 4: A side by side comparison of a model for a traditional SMT solver and the model for the SMT Solver we created in this project, respectively.

When the new solver discovers a conflict, it does so in fractions of a second; a complete solver may take significantly longer time to discover the same conflict. We incorporated the new theory solver into an SMT Solver that we developed in the Tarski system [17]. We designed our SMT Solver so that it calls our theory solver in an eager fashion, and calls the slow yet complete solver only when it has verified a complete solution. Figure 4 illustrates how the model of our SMT Solver differs from traditional SMT Solvers.

Implementing our SMT Solver required:

- Modifying and augmenting of algorithms to produce new algorithms that work within the SMT framework
- Implementing the new algorithms in a novel theory solver and incorporating them into an SMT solver we designed
- Conducting extensive experiments to measure performance improvements.

We built our new theory solver by augmenting and adapting algorithms presented in the paper BlackBox/WhiteBox Simplification [1] [2]. BlackBox/WhiteBox algorithms were not built with SMT solving in mind, leading to deficiencies which make them unsuitable for use as a theory solver. The most important deficiency is that the original algorithms do not compute explanations for theory checks, which the SAT solver requires in order to learn new clauses and perform backjumps. The bulk of the algorithm development part of this project focused on augmenting these algorithms so that they compute explanations.

We tested the new SMT Solver on a publicly available repository of benchmark problems called SMTLIB. We compared the performance of our new SMT solving architecture to a traditional SMT Solver which does not make use of our new algorithms and against a theory

solver alone. We hypothesized that we would see significant speed improvements for certain classes of problems. If our theory solver could quickly detect unsatisfiability, we predicted that the new architecture would yield significant performance gains. On the other hand, even for problems where our theory solver cannot make many contributions, we predicted it would not incur a significant performance cost. This is because it solves quickly compared to the complete theory solver. In the Results section, we explore the merit of our hypothesis.

4 BlackBox

The original BlackBox suite of algorithms as described in [1] takes a normalized formula such that each inequality is of the form $x_1^{e_1} * x_2^{e_2} * ... * x_n^{e_n} \sigma 0$. Each x^e is a single factor, and each factor is treated as a single variable. In other words, BlackBox deals with an abstraction of the original formula which retains information about the factor structure of the formula but loses information about the factors themselves. In this abstraction, BlackBox can solve the satisfiability problem and make deductions about the signs of factors in polynomial time.

Our augmentations give BlackBox the ability to return an explanation for both the satisfiability problem and for deductions it makes. This section presents two new algorithms, *SMICS2* and *MIDIE2*. *SMICS2* is a decision procedure which decides whether or not a formula is satisfiable in the BlackBox abstraction, and returns an explanation when a formula is deduced to be unsatisfiable. *MIDIE2* takes a formula in the BlackBox abstraction and makes sign deduction on factors in the formula.

4.1 Representing a Formula in BlackBox

BlackBox uses the abstract representation of a formula where each factor is a single variable in order to transform the formula into a matrix. It does so by mapping each inequality to a bit vector, and the entire formula to a rectangular bit matrix. In the matrix, the rows correspond to inequalities and the columns to specific factors. The matrix also reserves one column to represent the sign of the inequality. The column reserved for the sign bit appears with a 1 in a row if that inequality had the $<$ or $\leq$ sign, and 0 otherwise. The algorithm then divides the matrix into a left side and a right side. Factors present in at least one strict inequality receive a column in the left side of the matrix. If one of these factors appears in an inequality with an odd exponent, the algorithm sets the bit corresponding to that inequality and factor to 1. Otherwise, it sets the bit to a 0.

Factors which are strictly in non-strict inequalities receive a column in the right side of the matrix. If the factor has an even exponent in an inequality, the algorithm sets the bit corresponding to that inequality and factor to 2. If it has an odd exponent in an inequality, the corresponding bit gets set to 1. If it has a 0 exponent in an inequality, the corresponding bit gets set to 0.

For example, consider the following formula. f and g are strictly non-strict variables.

$$abc < 0 \wedge abc > 0 \wedge c^2de < 0 \wedge efg \geq 0 \wedge f^2g \leq 0$$

This formula is mapped to the following matrix, where the strictly non-strict side is in bold:

$$\begin{bmatrix} \sigma & a & b & c & d & e & \boldsymbol{f} & \boldsymbol{g} & inequality \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & abc < 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & abc > 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & c^2de < 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & efg \geq 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 1 & f^2g \leq 0 \end{bmatrix}$$

Figure 5: A matrix

We describe the process for creating a matrix for satisfiability computations formally in the algorithm *Inequality To Vector Mapping*.

Algorithm 1 Inequality to Vector Mapping (IVM)

Input: $L = \{l_1, l_2, ..., l_3\}$, a list of inequalities, with all variables numbered from 0 to $N-1$, where N is the number of variables

Output: M, the matrix representation of L, and VARLIST, the list of weak inequalities

1: Let B be a boolean array of size N with all values initialized to false.
2: Let VARLIST be an array of size N. Note that VARLIST[i] and $B[i]$ refer to the same variable.
3: **for all** $l_i \in L$ **do**
4: **for all** $v \in l_i$ **do**
5: **if** l is strict **then**
6: Let VARLIST[v] := VARLIST[v] $\cup \{i\}$
7: Let $B[i]$:= true
8: Let M be the $|L| \times (1 + |S|)$ binary array, initialized s.t. all $M[i][j] = 0$
9: Let ROWSUM be the $|L| \times |L|$ binary identity matrix
10: Let i_1, i_2, ..., i_k be the indices of the true entries in B Let j_1, j_2, ..., j_{n-k} be the indices of false entries in B Let B be the array $[i_1, ..., i_k, j_1, ..., j_{n-k}]$
11: **for all** l_i in L **do**
12: **if** the sign of l is $<$ or $\leq$ **then** set $M[i][0] := 1$
13: **else** set $M[i][0] := 0$
14: **for all** variables $v \in l$ **do** $M[i][B[v] + 1] := 1$ if the power of v is odd
15: Return M, VARLIST

Theorem 6 in [1] states that any inequality discovered by rowsum operations in the strict part of the matrix produced by the above procedure maps to an inequality implied by the formula. This theorem allows us to apply Gaussian elimination to the matrix to make interesting deductions on the formula.

4.2 SMICS2

[1] describes an algorithm for determining if a formula is unsatisfiable in BlackBox, called Strict Monomial Inequality Conjunction Satisfiability or SMICS. In the SMICS algorithm,

for reasons which we shall explain shortly, we drop the right hand side of the matrix and all rows which correspond to the null vector. The above matrix becomes:

$$\begin{bmatrix} \sigma & a & b & c & d & e \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 6: The same matrix as Figure 5, but representing the strict part of the formula

The algorithm then applies gaussian elimination to the matrix, and searches the result for a row of the form $\{1, 0, 0, ..., 0\}$. A row of this form indicates unsatisfiability, because it represents an inequality such as $x^2 * y^4 < 0$, which is unsatisfiable.

Recall that the right hand side of the matrix constraints strictly non-strict factors. Let $z \leq 0$. If we did not drop the right hand side of the matrix, a row of the form $\{1, 0, 0, ..., 0\ \}$ could represent an inequality such as $x^2 * y^4 * z^6 \leq 0$, which can be satisfied by setting $z = 0$. To avert this, we dismiss all rows which contain strictly non-strict factors. Also recall that we drop all rows which correspond to the null vector. These rows stand for inequalities such as $1 > 0$ or $x^2 > 0$, which are vacuously true and contain no useful information.

Our new variant of SMICS, called SMICS2, also determines satisfiability by applying gaussian elimination and searching for a row of the form $\{1, 0, 0, ..., 0\}$. If formula is determined to be unsatisfiable, it returns the set of inequalities which cause the formula to be unsatisfiable. Consider this formula:

$$xyz < 0 \wedge xyz > 0 \wedge xab > 0 \wedge yab > 0 \wedge yxbc > 0 \wedge zx > 0 \wedge yzaef > 0$$

The set of inequalities $xyz < 0 \wedge xyz > 0$, a subset of the original formula cannot both be true. Therefore, this formula is unsatisfiable. We refer to this subset of inequalities as an "UNSAT Core" of the formula, which explains why the formula is unsatisfiable. In this project, it is essential for the algorithm to return an UNSAT Core, since SMT solvers use the UNSAT Core to generate learned clauses.

We determine which inequalities are in the unstrengthened core by constructing a square identity matrix with as many rows as inequalities in the formula. Whenever we add two rows together in process of Gaussian Elimination, we add the same rows together in the identity. Therefore, once we have obtained a row of the form $\{0, 0, 0, ..., 1\}$, from the original matrix, the corresponding row in the identity matrix indicates which inequalities form the unstrengthened core. We describe this process formally in the algorithm *Matrix Satisfiability*.

Algorithm 2 Matrix Satisfiability (MS)

Input: M, the matrix representation of a conjunction of inequalities, and L, the original list of inequalities

Output: BBSAT or CORE, the unstrengthened UNSAT Core

1: Let MASK be a bitvector of size $|L|$. MASK$[i] = 1$ if $0 = M[i][k+1] = ... = M[i][N]$ and 0 otherwise.
2:
3: Let TRACEROW := [0, 1, ..., $|L| - 1$]
4: Let (M, TRACEROW) := *filter*(M, TRACEROW, MASK)
5: Do Gaussian Elimination on M. If Elimination takes the step row i += row j, do the same on ROWSUM.
6: **if** M does not contain a row of the form $[1, 0, 0, ..., 0]$, the UNSAT row **then**
7: return BBSAT
8: Let u be the index of the UNSAT row
9: Let (M, TRACEROW) := *filter*(M, TRACEROW, ROWSUM[u])
10: Let $ROWSUM$ be the identity matrix of size $|M| \times |M|$
11: Perform Gaussian elimination on M. If Elimination takes the step row i += row j, do the same on ROWSUM.
12: **if** M contains the null row **then** let u be the index of the null row and return to step 9
13: Let CORE := $\emptyset$
14: **for** i from 0 to $|M| - 1$ **do**
15: Let CORE := CORE $\cup$ TRACEROW[i]
16: Return CORE

One problem which we tackle in this new algorithm is the process of "strengthening" non-strict inequalities. Consider this formula:

$$xyz \leq 0 \wedge xyz \geq 0 \wedge xab > 0 \wedge yab > 0 \wedge yxbc > 0 \wedge zx > 0 \wedge yzaef > 0$$

This formula is identical to the previous formula, except that $xyz < 0$ and $xyz > 0$ are now $xyz \leq 0$ and $xyz \geq 0$. Since this subset of the formula can be satisfied by setting $xyz = 0$, we cannot give $xyz \leq 0 \wedge xyz \geq 0$ as a valid UNSAT Core. However, we can still use these inequalities in the UNSAT Core as long as we prove that each of the factors x, y, and z are not equal to 0. We do so by including some set of strict inequalities ($<$ or $>$) which contain those variables. In this problem, our UNSAT Core now becomes:

$$xyz \leq 0 \wedge xyz \geq 0 \wedge zx > 0 \wedge yab > 0$$

Algorithm 3 Inequality Strengthen (IS)

Input: CORE, the unstrengthened UNSAT Core of a formula, L, the original list of inequalities, and FACTLIST, the mapping from each factor to the inequalities which can strengthen it

Output: STCORE, the strengthened UNSAT Core of a formula

1: Create the lists WEAKFACTS, STRONGFACTS and REASONS
2: **for all** $l \in L$ **do**
3: **if** l is nonstrict **then** add all factors in l to WEAKFACTS
4: **else** add all factors in $L[t]$ to STRONGFACTS
5: Let WEAKVARS := WEAKFACTS − STRONGFACTS
6: Set ADDS := *ScoringFunction*(WEAKVARS, FACTLIST, L)
7: Let STCORE := CORE ∪ ADDS
8: Return CORE

Of course, we could have also chosen:

$$xyz < 0 \wedge xyz > 0 \wedge zx > 0 \wedge yzaef > 0$$

We prefer the former explanation over the latter since it is a smaller explanation. However, we lack a proper notion of what an optimal choice of strengthening inequalities consists of. Currently, we aim for a selection which contains the fewest extra factors with an algorithm which we believe is a good trade off between speed and optimality. However, we are not overly concerned with this part of determining the UNSAT Core. Experience shows that it typically takes very few extra inequalities to strengthen an explanation. In the algorithm *Scoring Function*, we use a greedy algorithm which scores each added inequality based on the number of factors it strengthens, and the number of excess factors it adds.

Algorithm 4 Scoring Function

Input: $F = f_1, f_2, ..., f_k$, the list of factors which must be strengthened, $L = l_1, l_2, ..., l_n$, the list of inequalities as tarski formulas, and FL, an array mapping factors to the set of inequalities in L which they appear in

Output: FIN, the list of inequalities which strengthen the variables in STR

```
1: Let SCORE be an array which stores the score of every inequality, with each element initialized to 0
2: Let PENALTY be a boolean array which indicates whether or not an inequality has been seen once, with all elements initialized to false
3: Let FIN be the empty list
4: for all f ∈ F do
5:     if FL = NULL then continue
6:     for all l ∈ FL[v] do SCORE[l] += 5
7:         if PENALTY[l] = false then
8:             for all f ∈ L[l] ∉ F do SCORE[l] −= 2
9:             PENALTY[l] := true
10: Let STRONG be the highest scored member of FL[v]
11: For all f ∈ L[STRONG], let FL[v] := NULL
12: FIN.add(STRONG)
13: Return FIN
```

Algorithm 5 SMICS2

Input: $L = \{l_1, l_2, ..., l_3\}$, a list of inequalities as Tarski formulas, with all factors numbered from 0 to $N - 1$, where N is the number of variables

Output: SAT or STCORE, a list of inequalities which form a set which is the unsat core of L

```
1: M, FACTLIST := InequaltiyToVectorMapping(L)
2: SAT, CORE := MatrixSatisfiability(M, L)
3: if SAT then return SAT
4: STCORE := InequalityStrengthen(CORE, FACTLIST)
5: Return STCORE
```

Filter is a helper function which is referenced throughout the above algorithms. It is used in order to remove rows from the matrix which do not contribute to the UNSAT Core. Rows which contribute to the deduction have the corresponding index in Q set to 1. After we apply the algorithm, M' contains only the rows of M whose indices we set to 1 in Q, and R' maps the rows of M' to the rows of M. Since the implementation of Filter is not an interesting topic for this project, we included none.

Algorithm 6 Filter

Inputs

1: Input M, an $m \times n$ bit matrix
2: Input R, an m dimensional integer vector
3: Input Q, an m dimensional bit vector

Output:

1: M', an $a \times n$ bit matrix, filtered s.t. $a \leq m$ and if $Q[j] = 1$, then $M[j]$ exists in M' and if $i_1 < i_2 < ... < i_a$ are the indices of the non-zero entries of Q, then row S of M' is row i_s of M
2: R', an a dimensional vector, filtered such that if $R'[s] = R[i_s]$

4.3 MIDIE2

[1] describes an algorithm for determining if a formula implies some equality, called Monomial Inequality Discover Implied Equation or MIDIE. We run our own variant of MIDIE (MIDIE2) when SMICS2 fails to determine the formula is unsatisfiable.

Recall that the algorithm *IVM* generates a matrix from a normalized formula. MIDIE2 also does not use the non-strict part of the matrix. It applies gaussian elimination and searches for a row of the form [x, 0, 0, ..., 1, 0, ... 0] where there is one unique 1 aside from the sign column. For the factor f, a row of this form indicates $f \geq 0$ or $f \leq 0$. If f is on the left side of the matrix, we may strengthen the deduction so that it indicates $f > 0$ or $f < 0$.

Rows which are already of the form [x, 0, 0, ..., 1, 0, ... 0] before Gaussian Elimination present a special case. We treat them differently from regular deductions and ignore them after Gaussian Elimination, since MIDIE2 has not actually deduced anything. However, if the unique column is in the left side of the matrix and the known sign is non-strict, we can strengthen the sign to the strict variant. The algorithm Preprocess handles these rows.

Algorithm 7 Preprocess

Input: M, the matrix representation of a conjunction of inequalities, and L, the original list of inequalities

Output: KNOWN, the indices of factors whose sign is already known, DEDS, the list of factors which we can strengthen without doing Gaussian Elimination, and DEPS, the dependency of each member in DEPS

1: **for** i from 0 to $|M| - 1$ **do**
2: **if** $M[i]$ is a row with a unique 1 not in the first column **then**
3: Add i to KNOWN
4: **if** i is in the left side of the matrix and $L[i]$ is nonstrict **then**
5: $\beta :=$ the strict variant of the sign of $L[i]$
6: Add ($L[i]$, β) to DEDS
7: ADD *InequalityStrengthen2*($L[i]$, L, $FACTLIST$) to DEPS
8: Return KNOWN, DEDS, DEPS

We augment MIDIE similarly to the way we augment SMICS. We append the identity matrix to the original matrix which we used to track row sum operations. The values of each row indicate which factors were used to make a deduction. Similarly, we must also strengthen each factor for which we deduce $f > 0$ or $f < 0$ rather than $f \geq 0$ or $f \leq 0$.

Algorithm 8 Matrix Deductions (MD)

Input: M, the strict matrix representation of a conjunction of inequalities, L, the original list of inequalities, and FACTLIST, the mapping from factors to inequalities which strengthen them

Output: DEDS, the list of deductions we can make where each deduction is a tuple consisting of a polynomial p, the deduced sign β, and a list of the dependencies of the deduction

```
1: STR := the empty list
2: KNOWN, DED, DEPS := Preprocess(M, L)
3: Do Gaussian Elimination on M, following the steps with ROWSUM.
4: for i from 0 to |M| − 1 do
5: for all rows r with a unique 1 not in the first column do str := false
6:     if r ∈ KNOWN then continue
7:     i := the column number of the unique 1 in r
8:     j := the row number of r
9:     dep := the empty list
10:    for all nonzero indices k ∈ ROWSUM[j] do
11:        Add L[k] to dep
12:        if M[k][j] = 1 and M[k][j] has no 1 in the right side of the matrix then
13:            str := true
14:    If M[j][0] is 0, β := >. Else, β := <
15:    if str then Add (L[j], β, dep) to DEDS
16:    else
17:        Add InequalityStrengthen2(L[j], L, FACTLIST) to dep
18:        Add (L[j], β, dep) to DEDS
19: Return DEDS
```

Lines 15-20 determine if a strengthening factor is already a part of the dependencies of the deduction. If there is no strengthening factor, we add a strengthening factor to the dependencies by calling *Inequality Strengthen 2*.

Algorithm 9 Inequality Strengthen 2 (IS2)

Input: WEAK, a factor which must be strengthened, L, the original formula, and FACTLIST, the mapping of factors to inequalities which can strengthen them
Output: STR, an inequality which strengthens WEAK

1: BEST := null, BESTSCORE := $-\infty$
2: **for all** ineq $\in$ FACTLIST[WEAK] **do**
3: num := the number of factors in INEQ
4: score := 7 + -2*num
5: **if** score > BESTSCORE **then** BEST := ineq, BESTSCORE := score
6: Return BEST

Algorithm 10 MIDIE2

Input: $L = \{l_1, l_2, ..., l_3\}$, a list of inequalities as Tarski formulas, with all factors numbered from 0 to $N-1$, where N is the number of factors
Output: DEDS, the list of deductions made by MIDIE2

1: M, FACTLIST := := $IVM(L)$
2: DEDS := $MD(M, L)$
3: Return DEDS

4.4 MinWtBasis2

MinWtBasis is another algorithm described in [1] which allows us to make deductions on a Matrix representation of a formula. However, while MIDIE and SMICS only apply to the strict part of a matrix, MinWtBasis allows us to make deductions on the nonstrict part of the matrix.

To augment MinWtBasis, we make use of a procedure called *ReduceExplain*. We provide no algorithm body, but define the input and output. We also use the definitions of the the "support of vector w," $S(w)$ as the set of indices from the non-strict part at which w is non-zero, and "weight of vector w" as $|S(w)|$.

Algorithm 11 ReduceExplain

Input: $L = \{l_1, l_2, ..., l_3\}$, a list of inequalities as Tarski formulas, with all factors numbered from 0 to $N-1$, where N is the number of factors, M, a matrix formed from the elements of L, and w, the bit vector to reduce. Optionally provide M_{EXP}, an explanation for each row in M
Output: w', which w is reduced by M, and *Exp*, the inequalities in L used to reduce w

Algorithm 12 MINWTBASIS2

Input: $L = \{l_1, l_2, ..., l_3\}$, a list of inequalities as Tarski formulas, with all factors numbered from 0 to $N-1$, where N is the number of factors, and M, the matrix representation of L

Output: DEDS, the list of deductions made by MINWTBASIS2

CANDIDATES := the empty list
DEDS := the empty list
w := a maximum weight element of B.
while $wt(w) \neq 0$ and $M \neq \{\}$ **do**
    $M := M - w$
    $M_{\leq} := \{b \in B | S(b) \subseteq S(w)\}$
    $M_{<} := \{b \in B | S(b) \subset S(w)\}$
    if a subset $T \subseteq B_{\leq}$ such that T implies $[1, 0, ..., 0] \quad [0, ..., 0]$ mod 2 **then** continue
    Do Gaussian elimination on M to put it in reduced row echelon form. Let M_{EXP} be a list which corresponds to the inequalities required to reduce each row
    w', Exp := *ReduceExplain*$(L, M_{\leq}, w, M_{EXP})$
    if w' or some row of M m equals $[1, 0, ..., 0] \quad [0, ..., 0]$ **then**
        Add $(2w + [1, 0, ..., 0] \quad [0, ..., 0])$, $M_{EXP}[m]$ or Exp to CANDIDATES
        remove from M any element with support the same as w
    else if $w' \neq [0, 0, ..., 0] \quad [0, ..., 0]$ **then**
        Add w', Exp to CANDIDATES
for all w, Exp in CANDIDATES **do**
    p := the atom represented by w
    sgn := $\leq$ if $w[0]$ is 1, and 0 otherwise
    if all entries in w are 0 or even **then** add (p, sgn, Exp) to DEDS and continue
    if sgn = $\leq$ **then** add (p, =, Exp) to DEDS
Return DEDS

5 WhiteBox

Recall that BlackBox algorithms operated in an abstraction where each factor became a blackbox and we did solving based on the factor structure of a formula. WhiteBox algorithms as described in [2] operate in a different abstraction, which allows us to open up the "blackboxes" based on the known sign information of each variable. In the WhiteBox abstraction, we do not assign a specific value to any variable. In whitebox, we only assign some relational operator in OP to each variable, where $OP = \{<, >, \leq, \geq, =, \neq, ?, X\}$. ? indicates we know nothing about the sign of a variable. X indicates that a variable is unsatisfiable. Once again, we can do satisfiability solving and make deductions about the signs of variables in this abstraction in polynomial time.

We augment WhiteBox algorithms so that they return a "maximally weak" explanation for how we may deduce some sign on a polynomial or a variable. Maximal weakness refers to which signs imply one another. For example, if we know $x < 0$, we automatically know that $x \leq 0$ and $x \neq 0$. If we have $x = 0$, we also have $x \geq 0$ and $x \leq 0$. Every sign also implies x ? 0, since ? sets no restricitons on the value of a variable. Maximal weakness is important in the overarching SMT Framework because it can reduce the amount of times we need to recompute explanations. For example, say that the fast theory solver discovers the unsatisfiability of some formula proposed by the SAT solver $F \wedge x = 0 \wedge x < 0$. Here, it can select either $x < 0 \wedge x = 0$ or $x \neq 0 \wedge x = 0$ as an explanation. Let us assume that it picks the weakest possible explanation, $x \neq 0 \wedge x = 0$. This explanation prevents the SAT Solver from proposing the formula $F \wedge x = 0 \wedge x > 0$ immediately after, and saves us the work of computing another explnation!

We divide the WhiteBox suite into three seperate classes of algorithms. The first algorithm is MonomialSign, which computes an explanation for the sign of some monomial. The next algorithm is PolynomialSign, which computes an explanation for the sign of some polynomial p by calling MonomialSign on each monomial in p. The final algorithm is DeduceSign, which makes deductions on a polynomial p by comparing two p and another polynomial q. Finally, we give a description for a complete WhiteBox algorithm which makes all possible WhiteBox deductions on some formula.

5.1 MonomialSign

The original WhiteBox MonomialSign algorithm as described in [2] determines the strongest possible sign on a Monomial based on known sign information of each variable. It takes as input a power product M consisting of variables and their exponents $x_i^{e_i}, ..., x_n^{e_n}$, as well as the signs $\alpha_i, ..., \alpha_n$ on each variable. It then iterates through each variable and sign, refining the sign of the power product on each iteration.

For the purposes of SMT solving, we take a somewhat different approach. We divide MonomialSign into several different algorithms, where each algorithm attempts to prove that the input implies a different sign $\beta \in \{<, >, \leq, \geq, \neq, =\}$ on M. If we cannot prove $M\beta 0$, then the algorithm simply returns "Fail". Otherwise, the algorithm returns a maximally weak certificate describing which signs imply $M\beta 0$.

Below, we describe each variant of the algorithm, accompanied by a proof that the algorithm provides a "maximally weak" explanation.

Algorithm 13 Provide Weakest Proof Strict (PWPS)

Input: Power Product $M = x_1^{e_1} * ... * x_n^{e_n}$ and $\alpha_1, ..., \alpha_n \in OP$, and $\beta \in \{<, >\}$, the sign of M that we desire to prove

Output: Fail when β cannot be implied by $\alpha_i, ..., \alpha_n$ or the sequence A $= \{\alpha'_1, ..., \alpha'_n\}$ such that A is as weak as possible and M with the signs of A $\implies M\beta 0$

1: Let CORRECT := *true*
2: **if** $\beta =<$ **then** let CORRECT := *false*
3: **for all** i from 0 to n **do**
4: **if** $\alpha_i \in \{?, \leq, \geq, =\}$ **then** return Fail
5: **if** e_i is even **then** set α'_i to $\neq$
6: **else if** α_i is $\neq$ **then** return Fail
7: **else** $\alpha'_i := \alpha_i$
8: **if** α_i is $<$ **then** let CORRECT := !CORRECT
9: **if** CORRECT = *true* **then** return α'_1, .., α'_n
10: **else** return Fail

Explanation: This algorithm operates by iterating through each variable $x_1, ..., x_n$ in the power product M and checking the corresponding α_i. It calls fail if $\alpha \in \{?, \leq, \geq, =\}$, since they do not imply $<$ or $>$. If e_i is odd, the algorithm cannot weaken $<$ or $>$. However, if e_i is even, the algorithm can weaken $<$ and $>$ to $\neq$, since a negative number raised to an even power is always positive. It keeps track of whether or not the $\alpha_1, .., alpha_n$ imply the correct sign of $<$ or $>$ by using a switch. It changes the switch every time it encounters $\alpha =<$.

Proof: Given $\alpha_i \implies M\beta 0$, where $\beta \in \{<, >\}$. Suppose α'_i is a component-wise weakening of α_i such that α'_i is "maximally weak" but $\alpha'_i \implies M\beta 0$ holds. Consider variable x_i and exponent e_i. If e_i is odd, then α'_i must be $\neq$, because all other sign conditions of α_i could be further weakened while still maintaining $\alpha'_i \implies M\beta 0$. If e_i is odd, then $\alpha_i \in \{<, >\}$. They cannot be further weakened, so therefore $\alpha'_i = \alpha_i$.

Algorithm 14 Provide Weakest Proof Non-strict (PWPN)

Input: Power Product $M = x_1^{e_1} * ... * x_n^{e_n}$ and $\alpha_1, ..., \alpha_n \in OP$, and $\beta \in \{\leq, \geq\}$, the sign of M that we desire to prove

Output: Fail when β cannot be implied by $\alpha_i, ..., \alpha_n$ or the sequence A = $\{\alpha'_1, ..., \alpha'_n\}$ such that A is as weak as possible and M with the signs of A $\implies M\beta 0$

1: Let CORRECT := *true*
2: **if** $\beta = \leq$ **then** let CORRECT := *false*
3: **for all** i from 0 to n **do**
4: **if** α_i is $=$ **then** return PWE(M, $\alpha_1, ..., \alpha_n$)
5: **if** e_i is even **then** set α'_i to ?
6: **else**
7: **if** $\alpha_i \in \{\neq, ?\}$ **then** return Fail
8: **if** $\alpha_i \in \{<, >\}$ **then** set α'_i to the non-strict variant of α_i
9: **if** α'_i is $\leq$ **then** let CORRECT := !CORRECT
10: **if** CORRECT = *true* **then** return α'_1, .., α'_n
11: **else** return Fail

Explanation: This algorithm operates very similarly to the previous algorithm, with a few additional twists. For one, it accepts $\alpha_i \in \{\leq, \geq, <, >\}$ and it weakens any $\alpha \in \{<>\}$ to $\leq$ or $\geq$, respectively. Additionally, it exhibits special behavior in the case where some α_i is $=$. In that case, it calls a different algorithm to generate an explanation through an entirely different procedure.

Proof: Given $\alpha_i \implies M\beta 0$, where $\beta \in \{\leq, \geq\}$ and for no j such that $e^j > 0$ do we have α_j is $=$. Suppose α'_i is a component-wise weakening of α_i such that α'_i is "maximally weak" but $\alpha'_i \implies M\beta 0$ holds. Consider variable x_i and exponent e_i. If e_i is odd, then α'_i must be ?, because all other sign conditions of α_i could be further weakened while still maintaining $\alpha'_i \implies M\beta 0$. If e_i is odd, then $\alpha_i \in \{\leq, <, \geq, >\}$. Since $<$ and $>$ imply $\leq$ and $\geq$ respectively, and the non-strict signs are weaker than the strict signs, $\alpha'_i \in \{\leq, \geq\}$.

Algorithm 15 Provide Weakest Proof Not Equal (PWPNE)

Input: Power Product $M = x_1^{e_1} * ... * x_n^{e_n}$ and $\alpha_1, ..., \alpha_n \in OP$

Output: Fail when $\neq$ cannot be implied by $\alpha_i, ..., \alpha_n$ or the sequence A = $\{\alpha'_1, ..., \alpha'_n\}$ such that A is as weak as possible and M with the signs of A $\implies M \neq 0$

1: **for all** i from 0 to n **do**
2: **if** $\alpha_i \in \{?, \leq, \geq, =\}$ **then** return Fail
3: **if** $\alpha_i \in \{<, >\}$ **then**
4: set α'_i to $\neq$
5: **else** $\alpha'_i := \alpha_i$
6: Return α'_1, .., α'_n

Explanation: This algorithm simply ensures that no $\alpha_i \in \{\leq, \geq, =, ?\}$ by iterating

through all $\alpha_1, ..., \alpha_n$.

Proof: Given $\alpha_i \implies M \neq 0$ suppose α_i' is a component-wise weakening of α_i such that α_i' is "maximally weak" but $\alpha_i' \implies M \neq 0$ holds. Since $\alpha_i \in \{<, >, \neq\}$ given our initial assumption, α_i' must be $\neq$ since otherwise α_i' is not maximally weak.

Algorithm 16 Provide Weakest Proof Equal (PWPEQ)

Input: Power Product $M = x_1^{e_1} * ... * x_n^{e_n}$ and $\alpha_1, ..., \alpha_n \in OP$
Output: Fail when = cannot be implied by $\alpha_i, ..., \alpha_n$ or the sequence A = $\{\alpha_1', ..., \alpha_n'\}$ such that A is as weak as possible and M with the signs of A $\implies M = 0$
Let ONE := *false*
1: **for all** i from 0 to n **do**
2: **if** $\alpha_i \in \{=\}$ and ONE = *false* **then** Let ONE := *true* and set α_i' to =
3: **else** set α_i' to ?
4: **if** ONE = *true* **then** return α_1', .., α_n'

Explanation: This algorithm selects one α_i to be = and weakens all others to ?, since any set of numbers multiplied by 0 is 0.

This algorithm presents a special problem in that we can choose any arbitrary α_i to be =. We have not yet investigated the possibility of suggesting to *PWPEQ* which variable should be set to 0.

5.2 PolynomialSign

The original WhiteBox PolynomialSign algorithm as described in [2] determines the sign of a polynomial based on the known sign information of each monomial. It takes as input a polynomial $p = a_1M_1 + a_2M_2 + ... + a_kM_k$, and the known sign information $\alpha_1, ..., \alpha_n$ on each variable in the polynomial. It outputs the sign information it learns about the polynomial.

We follow a similar tactic as we did for MonomialSign for augmenting PolynomialSign. We divide PolynomialSign into several different algorithms, each which try to provide the weakest proof possible for a target sign $\beta \in OP$, or fail if proving $p\beta 0$ is impossible.

Each variant of this algorithm functions by calling our augmented MonomialSign algorithms on each power product aM in the polynomial p. The algorithm then combines the signs returned from the call to the augmented MonomialSign with the already known sign information, resolving any conflicts by choosing the stronger of the two.

For example, say we determine that some Formula F is unsatisfiable because we deduce the polynomial $p = x^2yz^3 + wz$ is greater than 0. We then run the augmented PolynomialSign algorithm to determine an explanation for why $p > 0$. From the first monomial, we may deduce that $x > 0, y \geq 0$, and $z \geq 0$ and from the second $w > 0$ and $z > 0$. We resolve the conflict between $z \geq 0$ from the first monomial and $z > 0$ from the second by returning $z > 0$ in our final explanation. We call the combine algorithm in order to resolve these conflicts.

One interesting case where the augmented PolynomialSign algorithm exhibits special behavior is the case where any monomial in a polynomial is a square. In that case, we can deduce that that monomial must always have a sign that is at least as strong as $\geq$. For example, suppose that we discover some formula F is unsatisfiable because the polynomial

Algorithm 17 Combine

Input: Sequence A1 $= \alpha_1, ...\alpha_i$ and A2 $= \alpha'_1, .., \alpha'_i$
Output: Sequence A3 $= \alpha''_i, ..., \alpha''_i$ which is the strongest combination of A1 and A2.

$p = x^2y^2z^2 + a^2b^2c^2$ is greater than or equal to 0. Both of the monomials which make up p are always greater than or equal to 0, since they are perfect squares. Therefore, when we return an explanation for why $p \geq 0$, we do not need to return anything beyond the vacuous explanation that $\alpha_1?0, ..., \alpha_n?0$

Algorithm 18 Polynomial Provide Weakest Proof Strict (PPWPS)

Input: Polynomial $p = a_1M_1 + ... + a_kM_k$, where the M_i are power products over $x_1^{e_1}, ..., x_n^{e_n}$ and $\alpha_1, ..., \alpha_n \in OP$, and $\beta \in \{<, >\}$, the sign of M that we desire to prove
Output: Fail or the sequence A $= \{\alpha'_1, ..., \alpha'_n\}$ such that A is as weak as possible and p with the signs of A $\implies p\beta 0$

1: Let CANDIDATES := $\emptyset$
2: Let RES := $?_1, ..., ?_n$
3: **for all** i from 0 to k **do**
4: **if** PWPS(a_iM_i, β) $\neq$ Fail **then** CANDIDATES := CANDIDATES $\cup$ $\{a_iM_i\}$
5: **else if** PWPNS(a_iM_i, *nonstrict*(β)) $\neq$ Fail **then**
6: **if** β is $>$ and all e_i in a_iM_i are even **then** Continue
7: RES := *Combine*(RES, PWPNS(a_iM_i, β))
8: **else** return Fail
9: BESTSIGNS := *PolyScoringFun*(CANDIDATES)
10: **if** BESTSIGNS $\neq \emptyset$ **then**
11: RES := Combine1(RES, BESTSIGNS)
12: Return RES
13: **else** return Fail

Explanation: In order to prove $p > 0$ or $p < 0$, we require that all $a_iM_i \geq 0$ or $a_iM_i \leq 0$, respectively. We only require one monomial $a_iM_i > 0$ or $a_iM_i < 0$ in order to prove strictness. As long as one monomial is strict, the remaining monomials may be set to 0 and strictness still holds for the polynomials. We determine the best a_iM_i to set to strict by applying a scoring function to all strict a_iM_i.

Algorithm 19 PolynomialScoringFunction

Input: List EXPLAIN = $A_1, A_2, ..., A_k$, where each $A_1 = \{\alpha_1, ..., \alpha_n\}$ which stands for the result of some call to *PPWPS*, and Sequence $A^* = \{\alpha_1^*, ..., \alpha_n^*\}$ which is the current set of signs.

Output: $A' = \{\alpha_1', ..., \alpha_n'\}$, the weakest element of EXPLAIN

```
1: Let SMAP := {<: 7, >: 7, =: 0, ≤: 5, ≥: 5, ≠: 5, ≠: 4, ? : 0}
2: SCORE := +∞
3: for all A_i ∈ EXPLAIN do
4: score := 0
5:     for all α_i in A_i do tmpscore := SMAP[α_i] − SMAP[α_i^*]
6:         if tmpscore < 0 then tmpscore := 0
7:         score := score + tmpscore
8:     if score < SCORE then
9:         SCORE := score, A' := A_i
   return A'
```

Explanation: We use the Polynomial Scoring Function algorithm to decide which monomial to strengthen to $>$ or $<$ for calls to *PPWPS*. The algorithm examines the sign requirements of each candidate monomial, and assigns a score to each. The scoring process takes into account the already known sign requirements, and applies a stiffer penalty for establishing a new sign requirement than it does for strengthening an existing requirement. Similar to the situation for BlackBox, we don not have a definition of what the optimal choice for strengthening is. Also like before, this is ultimately a small sub-problem which does give cause for significant concern.

Algorithm 20 Polynomial Provide Weakest Proof Non-Strict (PPWPNS)

Input: Polynomial $p = a_1M_1 + ... + a_kM_k$, where the M_i are power products over $x_1^{e_1}, ..., x_n^{e_n}$ and $\alpha_1, ..., \alpha_n \in OP$, and $\beta \in \{<, >\}$, the sign of M that we desire to prove

Output: Fail or the sequence A = $\{\alpha_1', ..., \alpha_n'\}$ such that A is as weak as possible and p with the signs of A $\implies p\beta 0$

```
1: Let RES := ?_1, ..., ?_n
2: for all i from 0 to k do
3:     if PWPNS(a_iM_i, β) ≠ Fail then
4:         if all e_i in a_iM_i are even then continue
5:         else RES := Combine(RES, PWPNS(a_iM_i, β))
6:     else return Fail
7: Return RES
```

Explanation: In order to prove $p > 0$ or $p < 0$, we require that $a_iM_i \geq 0$ or $a_iM_i \leq 0$, respectively. Since we do not prove strictness in this algorithm, we do not need to apply a scoring function like in the previous problem.

Algorithm 21 Polynomial Provide Weakest Proof Not Equal (PPWPNE)

Input: Polynomial $p = a_1M_1 + ... + a_kM_k$, where the M_i are power products over $x_1^{e_1}, ..., x_n^{e_n}$ and $\alpha_1, ..., \alpha_n \in OP$

Output: Fail or the sequence A = $\{\alpha'_1, ..., \alpha'_n\}$ such that A is as weak as possible and p with the signs of A $\implies p \neq 0$

1: **if** p contains only one term **then** return PWPNE(p, $\alpha_1, ..., \alpha_n$)
2: Let RES := $?_1, ..., ?_n$
3: Let β := ?
4: $j := 0$
5: **while** β =? and $j \neq k$ **do**
6: $j := j + 1$
7: tmp := PWPS(a_jM_j, $\alpha_1, ..., \alpha_n$, >)
8: **if** tmp != FAIL **then**
9: RES := *Combine*(RES, tmp)
10: β :=>
11: break
12: tmp := PWPS(a_jM_j, $\alpha_1, ..., \alpha_n$, <)
13: **if** tmp != FAIL **then**
14: RES := *Combine*(RES, tmp)
15: β :=<
16: break
17: tmp := PWPE(a_jM_j, $\alpha_1, ..., \alpha_n$)
18: **if** tmp = FAIL **then** return Fail
19: RES := *Combine*(RES, tmp)
20: **if** $j = k$ **then** return RES
21: **while** $j \leq k$ **do**
22: tmp := PWPS(a_jM_j, $\alpha_1, ..., \alpha_n$, β)
23: **if** tmp = FAIL **then** tmp := PWPE(a_jM_j, $\alpha_1, ..., \alpha_n$)
24: **if** tmp = FAIL **then** return FAIL
25: RES := *Combine*(RES, tmp)
26: return RES

Explanation: In order to prove that some polynomial $p \neq 0$, we effectively prove that $p > 0$, $p < 0$, or $p = 0$. Ironically, if $p = a_1M_1 + a_2M_2$, and $a_1M_1 \neq 0$ and $a_2M_2 \neq 0$, we cannot prove $p \neq 0$. The possibility exists that $a_1M_1 = a_2M_2$, such that $p = 0$. Only if p is a monomial can p contain a monomial which is not equal to 0.

Algorithm 22 Polynomial Provide Weakest Proof Equal (PPWPE)

Input: Polynomial $p = a_1M_1 + ... + a_kM_k$, where the M_i are power products over $x_1^{e_1}, ..., x_n^{e_n}$ and $\alpha_1, ..., \alpha_n \in OP$

Output: Fail or the sequence A = $\{\alpha'_1, ..., \alpha'_n\}$ such that A is as weak as possible and p with the signs of A $\implies p = 0$

1: Let RES := $?_1, ..., ?_n$
2: **for all** i from 1 to k **do**
3: **if** PWPE(a_iM_i, β) $\neq$ Fail **then** RES := *Combine*(RES, PWPNS(a_iM_i, β))
4: **else** return Fail
5: return RES

Explanation: In order to prove that some polynomial $p = 0$, we must prove that each monomial in p is 0.

Finally, with all different cases covered, we may describe a complete PolynomialSign algorithm which returns an explanation. It is a simple controller which decides which variant of PolynomialSign to call, and returns the explanation.

Algorithm 23 PolynomialSign2

Input: Polynomial $p = a_1M_1 + ... + a_kM_k$, where the M_i are power products over $x_1^{e_1}, ..., x_n^{e_n}$, sequence $A = \alpha_1, ..., \alpha_n \in OP$, and β, the sign we desire to prove for p

Output: Fail or $\alpha'_1, ..., \alpha'_n \in OP \implies p\ \beta\ 0$

1: **if** β is $>$ or $<$ **then** return PPWPS(p, A)
2: **else if** β is $\geq$ or $\leq$ **then** return PPWPNS(p, A)
3: **else if** β is $\neq$ **then** return PPWPNE(p, A)
4: **else if** β is $=$ **then** return PPWPE(p, A)
5: **else** return Fail

5.3 DeduceSign

The original *DeduceSign* algorithms makes a deduction based on two different factors. Suppose we have p and q as factors in a formula, and we know $q > 0$. If for some positive constant t we deduce that $p + tq < 0$ based on $\alpha_1, ..., \alpha_n$, we may then also deduce that $p < 0$. Consider the following formula:

$$x > 0 \wedge y < 0 \wedge x - 2y < 0 \wedge y + 1 < 0$$

We can deduce that this formula is unsatisfiable if we set t to 2, p to x, and q to $y + 1$. $p + 2q = x - 2y + 2y + 2 = x + 2$. $x + 2 > 0$, but we achieved that by adding $2y + 2$, which is less than 0. That is a contradiction, and therefore the formula is unsatisfiable.

DeduceSign makes use of an algorithm calls *FindIntervals*, which provides intervals I_1 and I_2 such that for all $t \in I_1$, we can deduce $p + tq \geq 0$, and vice versa for $t \in I_2$. *FindIntervals* refines the intervals based on the signs of each monomial in p and q and the values of the

constant multiplicands. *DeduceSign* uses the returned intervals and the sign of q to make deductions on the sign of p.

Algorithm 24 FindIntervals2 (FI2)

Input: Polynomials $p = a_1M_1 + ... + a_kM_k$ and $q = b_1M_1 + ... + b_kM_k$, $\alpha_1, ..., \alpha_n \in OP$
Output: Q, a set of disjoint interval/relop pairs such that $(I, \beta) \in Q \implies \forall t \in I[p + tq\beta 0]$

1: Let $Q := ((-\infty, \infty), =)$
2: **for all** i from 1 to k **do**
3: ms := *MonomialSign*$(M_i; \alpha_1, ..., \alpha_n)$
4: **if** $b_i = 0$ **then**
5: cs :=> if $a_i > 0$, and < otherwise
6: L := {((−∞, ∞), tProd[cs][ms])}
7: **else**
8: **if** $b_i > 0$ **then** (sl, sm, sr) := $(<, =, >)$
9: **else**(sl, sm, sr) := $(>, =, <)$
10: L:= { $(-\infty, -\frac{a_i}{b_i})$, tProd[sl][ms]), $([-\frac{a_i}{b_i}], =), ((-\frac{a_i}{b_i}, \infty)$, tProd[sr][ms])) }
11: $Q' := \emptyset$
12: **for all** $I_1, \gamma_1 \in Q$ **do**
13: **for all** $I_2, \gamma_2 \in L$ **do**
14: $Q' := Q' \cup \{I_1 \cap I_2$, tSum$[\gamma_1][\gamma_2]\}$
15: Remove from Q' each q for which I_1 is empty or γ_1 =?
16: $Q := Q'$
17: return Q

FindIntervals2 appears to deviate significantly from the original *FindIntervals*. In reality, the two algorithms have the same input specifications and nearly the same output specifications, but use a different method to achieve output which gives the same information. *FindIntervals2* works with interval and relational operator pairs. It iterates through each monomial in p and q and produces interval/relational operator pairs for which we can determine $p + tq\ \beta\ 0$. It then combines each new pair with the pairs generated on previous iterations. At the end of the loop, it eliminates all pairs which contain an empty interval or the sign ?, and thus give us no useful information. Therefore, on each iteration, all existing pairs satisfy the loop invariant that all $t \in$ the interval I satisfy $p_i + tq_i\ \beta\ 0$ for all monomials so far considered, where β is the relational operator.

The algorithm uses the tables tProd and tSum to quickly make deductions on the result of multiplying and summing two factors with known sign information, respectively. For example, consider $x < 0 \wedge y < 0$. $x * y < 0$, while $x + y > 0$. We can make these two deductions with tProd and tSum by making two simple table lookups - tProd[<][<] is > and tSum[<][<] is <.

	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$	**?**
$<$	$>$	$=$	$\geq$	$<$	$\neq$	$\leq$	?
$=$	$=$	$=$	$=$	$=$	$=$	$=$	$=$
$\leq$	$\geq$	$=$	$\geq$	$\leq$	?	$\leq$	?
$>$	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$	?
$\neq$	$\neq$	$=$	?	$\neq$	$\neq$	?	?
$\geq$	$\leq$	$=$	$\leq$	$\geq$	?	$\geq$	?
?	?	$=$	?	?	?	?	?

	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$	**?**
$<$	$<$	$<$	$<$	?	?	?	?
$=$	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$	?
$\leq$	$<$	$\leq$	$\leq$	?	?	?	?
$>$	?	$>$	?	$>$	?	$>$	?
$\neq$	?	$\neq$	?	?	?	?	?
$\geq$	?	$\geq$	?	$>$	?	$\geq$	?
?	?	?	?	?	?	?	?

Figure 7: Tables for tProd and tSum

Our modification of *DeduceSign*, called *DeduceSignExplain*, calls *FindIntervals2* to find all possible intervals and relational operator pairs I from which we may deduce something interesting about p. *DeduceSignExplain* iterates through each pair and uses the known sign of q and the relational operator of I to deduce some sign information on p. If it the sign of q and the sign of I allow us to deduce something interesting, then we call *PolynomialSign* on $p + t * q$ with target the sign of I. The result of *PolynomialSign2* becomes the explanation for the new deduction. *DeduceSignExplain* returns the strongest possible deduction it can make on p, with the weakest possible explanation.

Algorithm 25 DeduceSignExplain

Input: Polynomials $p = a_1M_1 + ... + a_kM_k$ and $q = b_1M_1 + ... + b_kM_k$, β, the known sign of q, γ, the known sign of p, κ, the known sign of q, and sequence $A = \alpha_1, ..., \alpha_n \in OP$, the known signs of the variables in p and q.

Output: γ', the deduced sign on p, and $A' = \alpha'_1, ..., \alpha'_n \in OP$, which along with β is the weakest possible explanation for γ

1: $L :=$ *FindIntervals2*(p, q, A)
2: **if** L is empty **then** return (?, A)
3: $A'' :=$ the empty list
4: $\gamma' := \gamma$
5: **for all** I, $\kappa \in L$ **do**
6: $\quad \gamma^* :=$ tInterval$[t][\beta][\kappa]$
7: $\quad$ **if** γ^* is not at least as strong as γ' **then** continue
8: $\quad$ **if** γ^* is stronger than γ' **then** $\gamma' := \gamma^*$, $A'' :=$ the empty list
9: $\quad t :=$ the midpoint of I
10: $\quad$ Add *PolynomialSign2*($p + t * q$, κ) to A''
11: $A' :=$ *PolynomialScoringFunction*(A'', A)
12: return γ', A'

DeduceSignExplain uses a table to determine what sign it may deduce on p. This table is given below. The deduction first depends on the value of t for $p + t * q$, then on the sign

of q for the row, and then the sign of $p + t * q$ for the column.

	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$
$<$	$\leq$	?	?	?	?	?
$=$	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$
$\leq$	$<$	?	$\leq$	?	?	?
$>$	?	?	?	$\geq$	?	?
$\neq$	?	?	?	?	$=$	?
$\geq$	?	?	?	$>$	?	$\geq$

	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$
$<$	?	?	?	$\geq$	?	?
$=$	$<$	$=$	$\leq$	$>$	$\neq$	$\geq$
$\leq$	?	?	?	$>$	?	$\geq$
$>$	$\leq$	?	?	?	?	?
$\neq$	?	?	?	?	$=$	?
$\geq$	$<$	?	$\leq$	?	?	?

Figure 8: Table for tInterval for $>$ and $<$ respectively. q is represented by the rows, while $p + tq$ is represented by the columns. p is the output

6 Augmented BlackBox/WhiteBox

Now that we have described augmented version of all BlackBox and WhiteBox algorithms, we can describe a complete BlackBox/WhiteBox2 that combines both classes of deductions and provides explanations for each deduction it makes. We describe new variants of BlackBox and WhiteBox, and then combine them in a new DeduceAll algorithm, called DeduceAll2. Then, we describe how to find an explanation in terms of the inequalities in the original formula in the algorithm TraceBack. We combine DeduceAll2 and TraceBack to define a complete BlackBox/WhiteBox2.

We present an example of our implementation below as the Tarski command *bbwb*:

```
(bbwb [z + y^2 <= 0 /\ x z < 0 /\ (x^3 + 3 )(y + 7) <= 0 /\
       y - x >= 0 /\ a^2 b > 0])

UNSAT
Explanation: z + y^2 <= 0 /\ x - y <= 0 /\
            (y + 7)(x^3 + 3) <= 0 /\ (z)(x) < 0
Proof:
0: bb deduction: BlackBox UNSAT: y + 7 > 0 , x^3 + 3 > 0,
                                 (y + 7)(x^3 + 3) <= 0
1: poly sign: x^3 + 3 > 0 : x > 0
2: poly sign: y + 7 > 0 : y > 0
3: deduce sign: y > 0 : x - y <= 0 , x > 0
4: bb deduction: x > 0 : z < 0, x z < 0
5: bb deduction: z < 0 : z <= 0 ,
6: deduce sign: z <= 0 : z + y^2 <= 0
7: given atom: (y + 7)(x^3 + 3) <= 0:
8: given atom: x z  < 0:
9: given: x - y <= 0 :
```

```
All Deductions:
0: given z + y^2 <= 0 :
1: given x - y <= 0 :
2: given atom z + y^2 <= 0:
3: given atom x z < 0:
4: given atom (y + 7)(x^3 + 3) <= 0:
5: given atom x - y <= 0:
6: given atom a^2 b > 0:
7: deduce sign z <= 0 : z + y^2 <= 0
8: bb deduction z < 0 : z <= 0 , z < 0
9: bb deduction x > 0 : z < 0 , x z < 0
10: bb deduction b > 0: a^2 b > 0
10: deduce sign y > 0 : x - y <= 0 , x > 0
11: poly sign y + 7 > 0 : y > 0
12: poly sign x^3 + 3 > 0 : x > 0
13: bb deduction BlackBox UNSAT: y + 7 > 0 , x^3 + 3 > 0,
                                 (y + 7)(x^3 + 3) <= 0
```

This example demonstrates how combined BlackBox/WhiteBox2 generates proofs for UNSAT in terms of the original theory atoms. The line beginning with "Explanation" gives an UNSAT Core for this formula. In the proof section of the above output, we show each deduction that *bbwb* makes in order to determine UNSAT, along with the facts that each deduction is dependent on. Line 0 contains the actual deduction of UNSAT by the BlackBox algorithm SMICS2, which required the facts $y+7>0, x^3+3>0$, and $(y+7)(x^3+3)<=0$. In line 1, we deduced that $x^3+3>0$ by $x>0$ with the algorithm *PolynomialSign2*. That deduction was made possible by the deduction in line 4, where we deduced $x>0$ with the algorithm *MIDIE2* and the fact $z<0$. Line 4 was made possible by line 5, and line 5 was made possible by line 6, which was made possible by the given atom in line 8. Therefore, *bbwb* determines that $xz<0$ is a requirement to deduce UNSAT. We can follow the same process for the other requirements given in line 0 to examine how *bbwb* determines the UNSAT Core. Note that *bbwb* leaves the atom $a^2b>0$ out of the UNSAT Core, as it is not a requirement to deduce UNSAT.

6.1 DeduceAll2

DeduceAll2 describes how to use our augmented algorithms from the previous sections to perform formula simplification and satisfiability checking. Every time *DeduceAll2* makes a deduction, it adds the deduced sign and the dependencies of the deduction to a list of all deductions it made. This list of deductions is used by *TraceBack* in order to construct the explanation for each deduction, or the 'UNSAT Core in the case we deduce unsatisfiable. The algorithms presented below describe how to use BlackBox and WhiteBox algorithms in order to build this list.

One property of the BlackBox and WhiteBox algorithm suites is that deductions made

by one algorithm allow further deductions by another. For example, consider the formula $z \leq 0 \wedge x^3 + 3 \neq 0 \wedge xz < 0$. We can deduce that $x > 0$ by *MIDIE2*. Adding $x > 0$ to the formula allows us to deduce that $x^3 + 3 > 0$ by *PolynomialSign2*. Examples like this one raise the problem of determining what WhiteBox deductions we should attempt to make after learning the sign of some polynomial. The naive strategy is to simply try *DeduceSignExplain* on all polynomials p and q in a formula, and *PolynomialSign2* on all p in a formula. This is obviously inefficient, because many polynomials in the formula may not be impacted at all by the deduction we made (consider that it is useless to rerun *PolynomialSign2* on yz after we deduce $x < 0$). The algorithm *Requeue* selects which p and q we should attempt to make WhiteBox deductions on by requiring that they at least share one variable with the polynomial we made a deduction on.

Algorithm 26 Requeue

Input: Some polynomial p which we have discovered new information about

Output: Set varDeds, a set of pairs between a variable and a polynomial, set polySigns, a set of polynomials, and set polyDeds, a set of pairs of polynomials, where each element of all three sets indicate that it is possible we can make a new deduction

```
1: Let varDeds, polySigns, polyDeds := ∅
2: if p is a single variable monomial then
3:     for all q which contain p do
4:         Insert (p, q) into varDeds
5:         Insert (q) into polySigns
6:         Insert (q, p) into polyDeds
7:     for all q and r which contain p do
8:         Insert (q, r) into polyDeds
9: else
10:    for all q which share a variable with p do
11:        if q is a single variable monomial then insert (q, p) into varDeds
12:        Insert (p, q) and (q, p) into polyDeds
13: return (varDeds, polySigns, polyDeds)
```

Our complete WhiteBox2 algorithm uses the three sets generated by requeue to determine which procedures it should run on which polynomials. The order of priority is to run *DeduceSignExplain* on variable polynomial pairs, then to run *PolynomialSign2* on lone polynomials, and then to run *DeduceSignExplain* on polynomial pairs. The reason is for efficiency - deducing a sign on a single variable has the potential to affect many polynomials, and we can deduce more with *DeduceSignExplain* if we have accurate sign information on q. Any time the algorithm makes an interesting deduction, it calls *requeue* to determine what additional attempts it should make. *WhiteBox2* halts when there are no more potential candidates, i.e. varDeds, polySigns, and polyDeds are empty.

Algorithm 27 WhiteBox2

Input: Sets varDeds, polySigns, polyDeds, and Deductions, the list of all deductions made so far

Output: unsat and Deductions', Deductions with all possible Whitebox deductions appended

```
1: while varDeds, polySigns, and polyDeds are not empty do
2:     if varDeds is not empty then
3:         Remove an arbitrary (v, p) from varDeds
4:         (res, exp) := DeduceSignExplain(v, p, sign(v), sign(p), signs(v + vars(p)))
5:         res := res & sign(v)
6:         if res is not equivalent to sign(v) then
7:             sign(v) := res
8:             Add (v, res, exp) to Deductions
9:             if sign(v) = X then return (unsat := true, Deductions)
10:            else varDeds, polySigns, polyDeds ∪ requeue(v)
11:    else if polySigns is not empty then
12:        Remove an arbitrary (p) from polySigns
13:        res := PolynomialSign(p) & sign(v)
14:        if res is not equivalent to sign(v) then
15:            exp := PolynomialSign2(p, res)
16:            sign(p) := res
17:            Add (p, res, exp) to Deductions
18:            if sign(p) = X then return (unsat := true, Deductions)
19:            else varDeds, polySigns, polyDeds ∪ requeue(v)
20:    else
21:        Remove an arbitrary (p, q) from polyDeds
22:        (res, exp) := DeduceSignExplain(p, q, sign(p), sign(q), signs(p + vars(q)))
23:        res := res & sign(p)
24:        if res is not equivalent to sign(p) then
25:            sign(p) := res
26:            Add (p, res, exp) to Deductions
27:            if sign(p) = X then return (unsat := true, Deductions)
28:            else varDeds, polySigns, polyDeds ∪ requeue(v)
29: return (unsat := false, Deductions)
```

DeduceAll2 relies upon both BlackBox and WhiteBox deductions. Below, we present our *BlackBox2* algorithms. Note that *BlackBox2* does not depend on the sets varDeds, polySigns, and polyDeds. This is because the entire formula and all known sign information is modeled as a matrix such that the algorithm always operates on the entire formula. However, a BlackBox deduction made with MIDIE2 opens up the possibility of further WhiteBox deductions. Therefore, BlackBox2 does output the sets varDeds, polySigns, and polyDeds to inform WhiteBox which deductions it should attempt to make. Outputting these sets naturally lends to the idea that we can call *BlackBox2* and *WhiteBox2* in alternation in order to expand the scope of the deductions we can make.

Our algorithm is simple. First, it runs SMICS2 to check if a formula is unsatisfiable. If we cannot deduce unsatisfiable, then we run MIDIE2 to make all possible BlackBox deductions.

Algorithm 28 BlackBox2

Input: L, a conjunction of inequalities as Tarski Formulas, and Deductions, a list of deductions made so far

Output: *unsat*, Deductions', Deductions with all possible deductions appended, and sets varDeds, polySigns, polyDeds

```
1: Let varDeds, polySigns, polyDeds := ∅
2: (unsat, EXP) := SMICS2(L)
3: if unsat then
4:     Add EXP to Deductions
5:     return (unsat, Deductions)
6: Deductions2 := MIDIE2(L) ∪ MinWtBasis2(L, Matrix(L))
7: for all (p, β, exp) in Deductions2 do
8:     Add (p, β, exp) to Deductions
9:     sign(p) := β & sign(p)
10:    if sign(p) = X then return (unsat:= true, Deductions)
11:    else varDeds, polySigns, polyDeds ∪ requeue(p)
12: return (unsat := false, Deductions, varDeds, polySigns, polyDeds)
```

Before we can describe a complete *DeduceAll2* algorithm, we find it necessary to do additional bookkeeping. *Preprocess Formula* initializes our list of deductions so that it contains all known information explicit in the formula. This step is necessary for determining the UNSAT Core, which we describe in the next section. Additionally, it also initializes varDeds, polySigns, and polyDeds so that we attempt to make all possible WhiteBox deductions on all combinations of p and q for the formula for the first run only.

Algorithm 29 Preprocess Formula

Input: L, a conjunction of inequalities as Tarski Formulas
Output: Givens, the list of all given facts in L, sets varDeds, polySigns, and polyDeds

```
1: Givens := the empty list of Deductions
2: for all inequalities l_i with sign β_i in L do
3:     Add (l_i, β_i, null) to Deductions
4: varDeds, polySigns, polyDeds := ∅
5: deds = GetGivens(L)
6: for all Variables v in L do
7:     for all Polynomials p where sign(p) is not ? and p contains v do
8:         Insert (v, p) into varDeds
9:         Insert (p, v) into polyDeds
10: for all Polynomials p in L do
11:     Insert p into polySigns
12:     for all polynomials q in L which have a variable in common with p do
13:         Insert(p, q) into polyDeds
14: return (Givens, varDeds, polySigns, polyDeds)
```

Finally, we may describe a complete *DeduceAll2* algorithm. We determine all given information explicit in the target formula, and save it to our list of deductions. We run BlackBox once, and then WhiteBox once on all p and q. The first run of BlackBox and WhiteBox is guaranteed. Afterwards, we run BlackBox and WhiteBox as long as the previous round deduced something interesting. This process repeats until we determine UNSAT or nothing new can be deduced. In both cases we return the list of all deductions made.

Algorithm 30 DeduceAll2

Input: L, a conjunction of inequalities as Tarski Formulas
Output: *unsat* and Deductions, the list of all deductions made

```
1: (Deductions, varDeds, polySigns, polyDeds) := Preprocess(L)
2: (unsat, Deductions) := BlackBox2(L, Deductions)
3: if unsat then return (unsat, Deductions)
4: (unsat, Deductions) := WhiteBox2(Deductions, varDeds, polySigns, polyDeds)
5: if unsat then return (unsat, Deductions)
6: varDeds, polySigns, polyDeds := ∅
7: If any deductions were learned by WhiteBox2, madeDeds := true
8: while madeDeds = true do
9:     (unsat, Deductions, varDeds, polySigns, polyDeds) := BlackBox2(L, Deductions)
10:    if unsat then return (unsat, Deductions)
11:    If varDeds, polySigns, polyDeds all = ∅, break
12:    (unsat, Deductions) := WhiteBox2(Deductions, varDeds, polySigns, polyDeds)
13:    if unsat then return (unsat, Deductions)
14:    varDeds, polySigns, polyDeds := ∅
15:    If any deductions were learned by WhiteBox2, madeDeds := true
16: return (unsat := false, L)
```

6.2 Traceback

In the previous section, we treated our list of deductions as a generic list of tuples. That belies the more sophisticated data structure we use in order to retrieve the unsat core. Whenever we add a new deduction to the list, we tag the deduction with the indices of the facts which it depends on. Consider the following list of deductions:

1. $x^7 + 13y^2 \leq 0$, null
2. $5z - 500 > 0$, null
3. $10z - 10000 < 0$, null
4. $3y - z \leq 0$, null
5. $4y - x \geq 0$, null
6. $x - 120 \leq 0$, null
7. $x - 100 \geq 0$, null
8. $x \geq 0$, 7
9. $z > 0$, 2
10. $y > 0$, 8, 5
11. $x^7 + 13y^2 > 0$, *UNSAT*, 10, 7, 6

We mark deductions as null if they are facts given in the original formula. Otherwise, we mark them with numbers corresponding to the indices of the facts used to create them. For instance, we deduce that $y > 0$ because $4y - x \geq 0 \wedge x \geq 0$.

We retrieve the unsat core by doing a priority queue based search through the dependencies of each deduction. When we encounter a deduction with no dependencies (i.e., a given fact), we add it to our unsat core. If not, we add all the dependencies to our queue. This procedure is effectively a graph traversal, where the container for our graph is a list.

Following this procedure yields $x^7 + 13y^2 \leq 0 \wedge x - 100 \geq 0 \wedge 4y - z \geq 0$ as the unsat core.

We determine which indices must be added to a deduction by use of a map. The map, which we call polyToIndex, maps a polynomial p to the last deduction which updated the sign of p. We determine each required index by looking up the index with polyToIndex.

One problem which we face is saving the use of weakened signs in order to make a deduction. For example, consider the formula fragment $xz \geq 0 \wedge x > 0 \wedge z > 0$. We may deduce $xz > 0$ with explanation $x > 0 \wedge z \geq 0$. Similarly to how we mark each deduction with the indices of each of its dependencies, we also mark the deduction with $z \geq 0$ to mark that we have relaxed the requirement on the sign of z for our proof. When we perform the graph traversal, every time we encounter $z > 0$, we check the deduction which depends on $z > 0$ to see if it relaxes $z > 0$ to $z \geq 0$. If all deductions only require $z \geq 0$, then we may add $z \geq 0$ to the unsat core rather than $z > 0$.

Algorithm 31 Add Deduction

Input: A deduction d of the form (p, β, reasons) to add to the deduction list L
Output: Update polyToIndex and add d to the list with the indices of all dependencies

```
1: weak := the empty list
2: indices := the empty list
3: for all r_i in reasons do
4:     idx := polyToIndex[r_i]
5:     if r_i is weaker than L[idx] then
6:         Add (r_i, sign(r_i)) to weak
7:     Add idx to indices
8: polyToIndex[p] = the size of L
9: Add weak, indices to d
10: Add d to l
```

Algorithm 32 Traceback

Input: An index valid i in the deduction list L, where each entry is of the form (p, β, reasons, indices, weak)

Output: Formula R, a subset of the original formula which allows us to deduce $L[i]$

```
1: R := the empty list
2: PQ := an empty priority queue of pairs. PQ's top is the largest pair, which is determined
   by the first element in the pair and then the second.
3: seen := a boolean array of size |L| with all entries set to false
4: seen[i] := true
5: (p, β, reasons, indices, weak) := L[i]
6: weaksgn := ?
7: for all idx_n in indices do
8:     Insert (idx_n, i) into PQ
9: while PQ is not empty do
10:    (i, j) = PQ.top()
11:    PQ.dequeue()
12:    if seen[i] then continue
13:    (p, β, reasons, indices, weak) := L[i]
14:    if indices is empty then
15:        sgn := sign(L[j])
16:        if sgn = β then
17:            add (p, β) to R
18:            seen[i] := true
19:        else if sgn is stronger then weaksgn then
20:            weaksgn := sgn
21:        (i_2, j_2) := PQ.top()
22:        if i_2 ≠ i then
23:            add (p, weaksgn) to R
24:            seen[i] := true
25:    else
26:        seen[i] := true
27:        for all idx_n in indices do
28:            Insert (idx_n, i) into PQ
29: return R
```

6.3 Future Improvements

We have identified several improvements that we might make in order to make our algorithms more competitive. Some optimizations include:

1. Adding additionally types of deductions such as sums of squares
2. Adding another algorithm to do formula simplification with explanations

3. Designing an incremental version of *DeduceAll2*, so that each time we call the algorithm we do not need to recompute work already done

7 Implementation

In this section we discuss the implementation of the augmented algorithms that constitute the primary contribution of this paper. Implementation of the algorithms is a key goal of this project, and we needed to integrate them into an SMT solver as a proof of concept and to conduct experiments. We discuss the implementation of *BlackBox2*, *WhiteBox2*, and *DeduceAll2* first. Then, we describe our SMT Solver, and some of the major challenges we needed to overcome in doing so.

We implemented the algorithms discussed in this paper in the C++ language through the Tarski program [17]. We implemented *BlackBox2* as the command `bbsat` in Tarski. `bbsat` determines whether or not a formula is BlackBox satisfiable, e.g. it cannot determine satisfiable. It returns the UNSAT Core if it is unsatisfiable, or any deductions it can make if it is BlackBox satisfiable.

```
(bbsat [ a^2 b >= 0 /\ b < 0 /\ a c >= 0 /\ a b < 0])
(UNSAT:sym [b < 0 /\ (a)(b) < 0 /\ (a)^2(b) >= 0]:tar )
```

We implemented *WhiteBox2* as the command `wbsat` in Tarski. WhiteBox is similar to `bbsat` in that it determines whether or not a formula is WhiteBox satisfiable, and presents an UNSAT Core or any deductions made. It also presents a proof for UNSAT using the algorithms discussed in section 9.

```
(wbsat [x - 1 > 0 /\ y^2 + x^2 - 1 > 0 /\ y < 0 /\ y - x > 0])
(UNSAT:sym [x - 1 > 0 /\ y - x > 0 /\ y < 0]:tar )
```

Finally, we also implemented *DeduceAll2* as `bbwb` in Tarski. Like `wbsat`, it also presents a proof for UNSAT.

To test our hypothesis that our algorithms will at best substantially cut solving time and at worst cause a small penalty to solving time, we implemented an SMT Solver in Tarski. Recall that a traditional SMT Solver has 3 components: (1) the Propositional Satisfiability solver, which handles the logical structure of the formula, (2) the theory solver, which analyzes the satisfiability of a conjunction in a given theory, and (3) the SMT Solver itself, which handles the communication between (1) and (2). Also recall that the proposed model for this project adds an additional component: a fast theory solver in between the complete theory solver and the SMT solver . We used our implementation of *DeduceAll2* as the fast theory solver in our proposed model for SMT Solvers. We used a solver called QEPCAD (Quantifier Elimination by Cylindrical Algebraic Decomposition) [15] as the slow but complete solver. QEPCAD is not the optimal choice for SMT solving, as it is designed for quantifier elimination vice satisfiability solving. However, we made the choice to use QEPCAD because Tarski already has an implementation for calling QEPCAD, and it suffices for the task of verifying the satisfiability of a formula. We also modified a propositional satisfiability solver called MiniSat [18] to handle propositional logic in the context of our SMT solver. MiniSat is a common choice for many SMT Solvers , since it is a well-documented and high-performance propositional solver. This required non-trivial modifications to the MiniSat source code. We defined an interface for MiniSat which allows users to plug in

arbitrary theory solvers, and we wrote code which allows those theory solvers to teach Minisat new clauses.

Our SMT Solver performs three non-trivial critical functions. The first is to generate the propositional satisfiability problem for MiniSat to solve. Secondly, we translate the propositional stack which MiniSat generates into theory assumptions. In other words, we transform MiniSat variables such as 1 and 2 into the facts they stand for, such as $x > 0$. Finally, we process the results of `bbwb` and QEPCAD. If either operation generates an UNSAT core, we translate the core from theory facts back into MiniSat variables, and use them to teach MiniSat a learned clause.

7.1 Future Improvements

While this is a complete implementation, we have identified several improvements that we might make in order to make our solver more competitive with other tools, such as z3. Some optimizations include:

- Integrating NUCAD, a variant of QEPCAD, into Tarski and into this solver
- Implementing the algorithmic improvements mentioned in the *DeduceAll2* section

8 Experimentation And Results

We conducted experiments on a set of problems from SMTLIB, a repository of SMT benchmark problems. SMTLIB contains benchmark problems for many different theories, including problems used as benchmarks in SMT Solver competitions. We tested three different solvers on these problems:

1. our proposed solver, which uses *DeduceAll2* in an eager fashion and QEPCAD in a lazy fashion,
2. a lazy SMT Solver with only QEPCAD and MiniSat, and
3. a solver which calls QEPCAD directly without the SMT framework[1].

The SMT Solver with BlackBox/WhiteBox algorithms is our proposed model for SMT Solvers. The second SMT Solver is a traditional model of SMT Solvers designed for difficult theories, such as the theory which is the subject of this paper. It also sets a "baseline" for measuring how much our algorithms improve the performance of our solver compared to applying QEPCAD alone, which ignores the SMT framework entirely. We predicted that the solver with BlackBox/WhiteBox algorithms will have significantly improved performance for unsatisfiable problems, and will exhibit a small penalty to performance for satisfiable problems where it cannot contribute many deductions of UNSAT.

[1] QEPCAD is unlike many other theory solvers in that it can solve problems which are not pure conjunctions, albeit with varying efficiency. That characteristic of QEPCAD allows us to make this comparison. Recall that the SMT framework only requires that theory solvers are able to handle pure conjunctions.

We selected problems in the QF_NRA (Quantifier Free Non Real Arithmetic) library of SMTLIB, which is the theory this project pertains to. We filtered out problems which could not be represented by the Tarski system. We then divided the problems by satisfiability, which is a common division in the Satisfiability community. After applying these filters and divisions, we ended up with 4475 files which represented satisfiable problems, and 2930 problems which represented unsatisfiable problems.

We implemented Tarski commands for each solver, and then compared solve times by running the commands on all files via a Python script. Each command used the same procedure to read a file and load a formula, and only diverged once it was time to actually solve the problem. Additionally, we also applied "level 1 normalization" to the problem before we called our new code. Level 1 normalization changes atoms such as $-1(x^2) < 0$ to $x^2 > 0$, and can deduce false if an atom such as $x^2 < 0$ exists in a conjunction in a formula, which makes it easier to write code to handle logical formulas. Each problem also gave QEPCAD a timeout of 200 seconds, at which point program execution would terminate and the problem would be declared unsolvable for that solver.

8.1 Unsatisfiable Problems

For unsatisfiable problems, we see great benefit to applying BlackBox/WhiteBox algorithms as an intermediate theory solver.

One way of representing the observed performance through Figure 9. Figure 9 is a graph of time to number of files solved, where files are organized in time-ascending order for each strategy. Therefore, the order of files is different for all three strategies. In this representation, a curve which is to the left of another curve indicates superior speed to that other curve. A curve which is above another curve indicates the first curve represents a solver which solves more problems. This is a standard way of representing the performance of SMT solvers in this field.

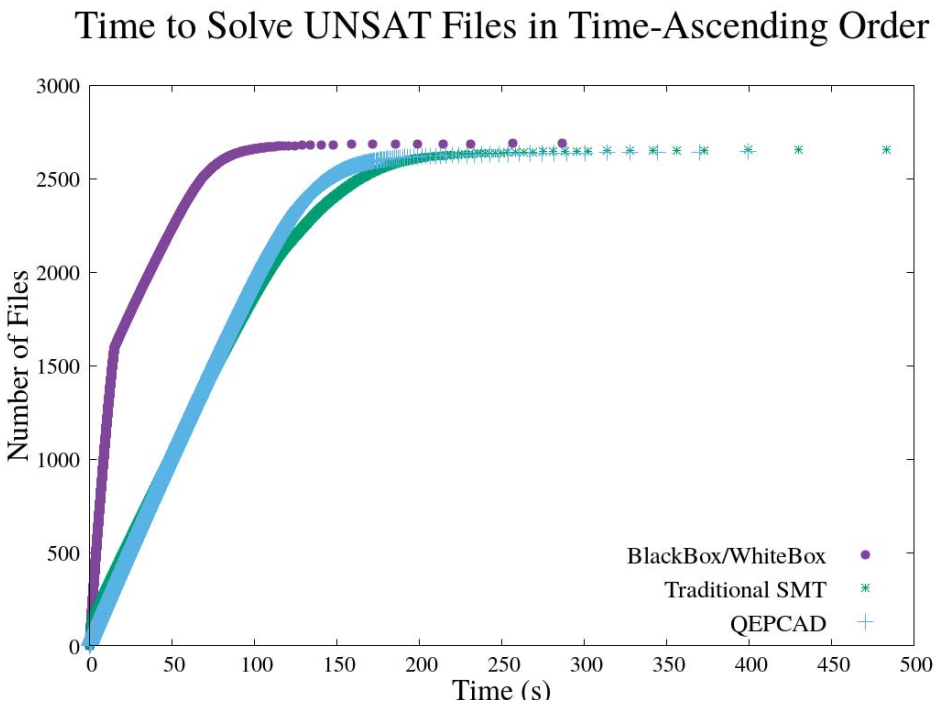

Figure 9: Total time to solve all unsatisfiable files. Of particular interest is the slope for the three different strategies up to 1500 files. Note that the BlackBox/WhiteBox SMT Solver is much steeper than the other two strategies.

In Figure 9, we see that the curve for the BlackBox/WhiteBox SMT solver has a much greater slope than the standard SMT Solver and QEPCAD alone for approximately 1500 files. This indicates greatly superior performance on those problems. Those problems are problems which BlackBox/WhiteBox can solve immediately without calling the complete theory solver, QEPCAD. After the 1500 problem mark, we see that the slopes for all three solvers become roughly equal. These are problems where generally BlackBox/WhiteBox cannot filter out all solutions by itself, and the solution requires a call to QEPCAD. In these problems we still see some small improvements since BlackBox/WhiteBox may filter out some solutions, but not to the same extent. Finally, after approximately 2500 files all three curves become almost parallel to the x-axis. These are problems which BlackBox/WhiteBox cannot solve at all, and are very difficult for QEPCAD to solve. However, note that this section of the curve is smaller for BlackBox/WhiteBox than it is for the other two strategies. This indicates that some files which are very difficult for QEPCAD to solve are solved immediately by BlackBox/WhiteBox.

Finally, note that the curve for the BlackBox/WhiteBox SMT solver ends at a point above and to the left of the curves for the other two solvers. This indicates that the BlackBox/WhiteBox SMT solver solves more files than the other two strategies, and does so significantly quicker than the other two strategies. We see that it solves 44 files which the traditional solver cannot solve at all, and 66 files which QEPCAD alone cannot solve at all.

We also represented the performance of our solver using the histogram displayed in Figure 10. In this Figure, we give comparisons of the performances of the different solvers on each file as a histogram. Each file is organized into a bin based on how fast or slow the SMT

Solver with BlackBox/WhiteBox algorithms solves the problem compared to the other two strategies. We organized the bins by taking the base two logarithm of the quotient of the time taken by the compared solver over the time taken by the solver with BlackBox/WhiteBox.

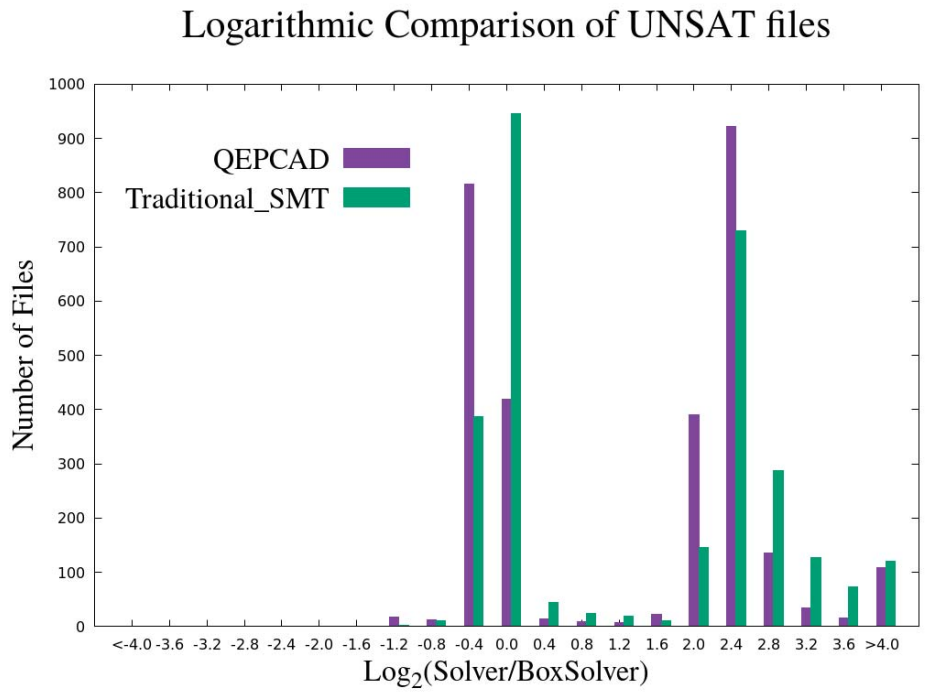

Figure 10: Comparisons for unsatisfiable files. Notice that most files are placed in the right side of the histogram, indicating the BlackBox/WhiteBox SMT solver performs better for those files than the other two strategies we tested.

In Figure 10, we see that there are over 100 files for each comparison in the >4.0 bin. This indicates that for over 100 files, either the BlackBox/WhiteBox SMT solver was able to solve the problem and the other solver could not, or that it was at least 16 times faster! On the other hand, we see few problems on the left side of the histogram, and no problem is sorted into a bin which indicates that the BlackBox/WhiteBox SMT solver is significantly slower than another strategy.

Therefore, we determine that the BlackBox/WhiteBox SMT solver is a much better choice for unsatisfiable problems than the traditional solver and QEPCAD alone, and thus that BlackBox/WhiteBox algorithms have significant effect on the difficulty of solving unsatisfiable problems.

8.2 Satisfiable Problems

For satisfiable problems, we see a benefit to the application of BlackBox/WhiteBox algorithms, but not to the same extent as unsatisfiable problems.

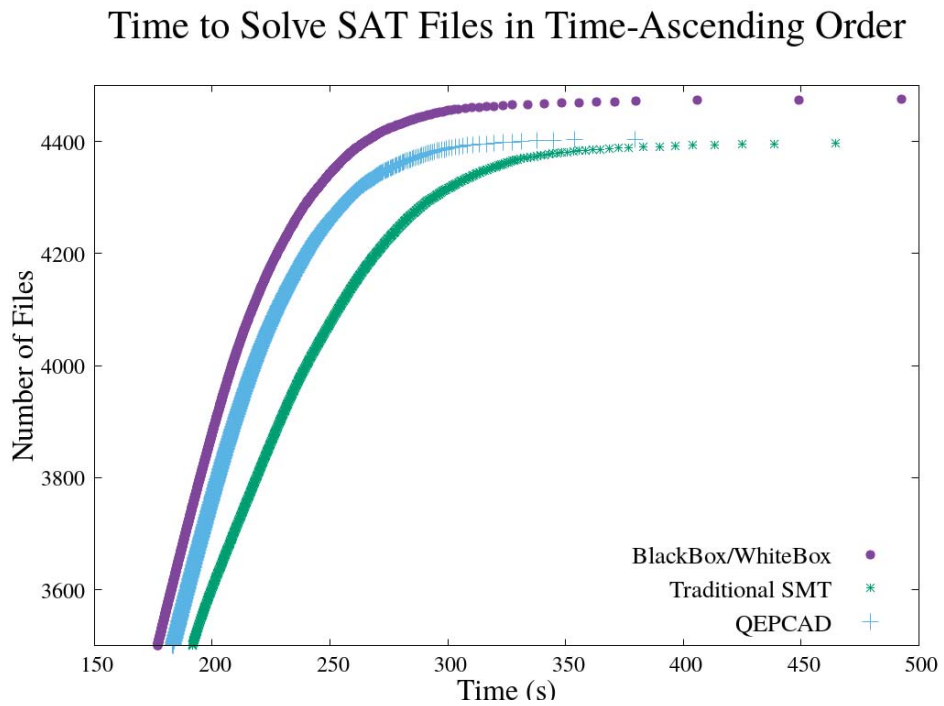

Figure 11: Total time to solve all satisfiable files. The plot is zoomed in to highlight the more interesting part of the comparison. The full plot shows the same behavior.

In Figure 11, we see that the BlackBox/WhiteBox SMT solver outperforms the other two solvers, but not by a significant amount. We also see that it solves more files than the other two solvers. In fact, it solved 4475 files, while the traditional SMT solver solved 4399 files and QEPCAD alone solved 4408 files.

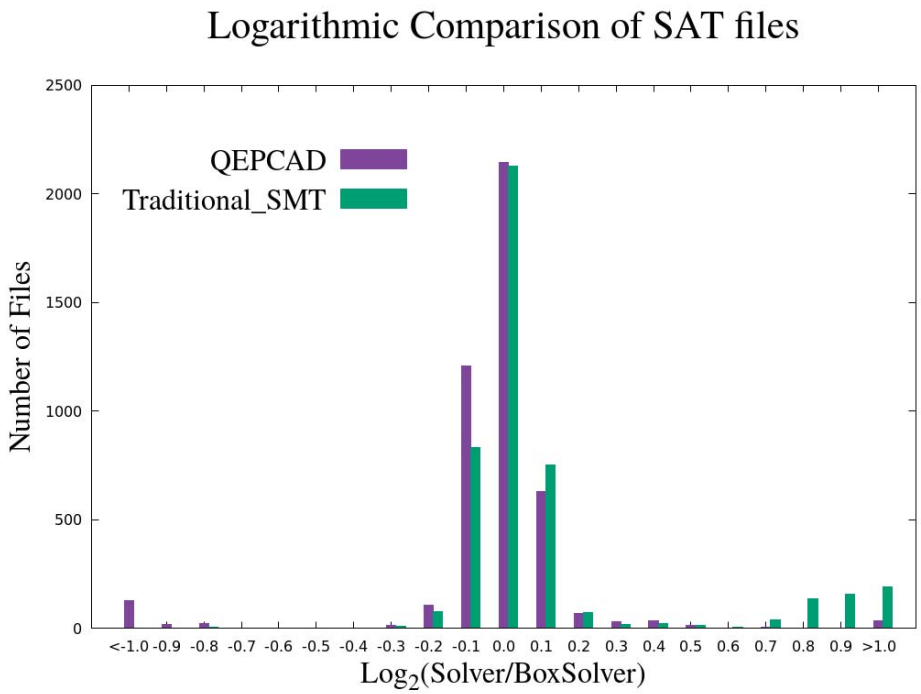

Figure 12: Comparisons for satisfiable files. Note that most files are clustered in the center of the graph, indicating that there is little change in solve times.

In Figure 12, we can see that on a per-file comparison the comparatively worse performance of the SMT solver with BlackBox/WhiteBox. In this chart, very few problems are more than twice as slow or fast for any given strategy. Some problems are solvable only by the SMT Solver with BlackBox/WhiteBox, but also some problems are unsolvable by it which are solvable by other strategies.

We can explain the reduced performance of the solver with BlackBox/WhiteBox algorithms by modeling SMT problems as a search tree. Every time the propositional solver presents a solution, it travels down one branch of the search tree. This end of the search tree may be satisfiable or unsatisfiable. In the case of unsatisfiable problems, all branches the propositional solver may travel down end in unsatisfiable, and therefore BlackBox/WhiteBox may reject it and contribute to solver performance. In satisfiable problems, we may have some ends which are unsatisfiable with at least one satisfiable branch, or no branches at all which are unsatisfiable. BlackBox/WhiteBox as is currently implemented can only contribute when the solver travels down an unsatisfiable branch. Therefore, BlackBox/WhiteBox has less opportunities to contribute.

We found that the cases where our proposed SMT solver was significantly outperformed by QEPCAD tended to be formulas with long conjunctions and simple disjunctions. In these cases, the satisfiability solver happened to make the wrong "guess" on which member of the disjunction to add to the conjunction and solve first. That bad guess changed the problem from one which is easy to solve to a problem which causes QEPCAD to take a significant amount of time to solve or, worse, time out.

9 Conclusion

We found that our results are consistent with our hypothesis that our new SMT architecture would have significantly improved performance at best and slightly penalized performance at worse. BlackBox/WhiteBox algorithms provide a significant advantage over a full theory solver when they can determine UNSAT. At the same time, since BlackBox/WhiteBox is guaranteed to run in polynomial time, a bad call to BlackBox/WhiteBox doesn't have a significant effect on solver efficiency. In fact, the cost of applying BlackBox/WhiteBox algorithms is small enough that we can still see some benefit for satisfiable problems when they cannot contribute as much. We saw that examples where our solver took a significant penalty to performance had little to do with BlackBox/WhiteBox and more to do with the slowness of the full theory solver for a particular problem.

As previously stated, we have several strategies for reducing solver time even further in the future. In particular, we predict that adding simplification with explanation to our augmented algorithms will have a significant impact on solver time for satisfiable examples. By integrating this improvement we will be able to entirely eliminate some constraints which makes problems difficult for complete theory solvers to produce solutions for. We anticipate that simplification will let us see performance improvements which are not quite as dramatic as in the unsatisfiable case, but much better than our current results.

References

[1] Christopher W. Brown. Fast simplifications for tarski formulas. In *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*, ISSAC '09, pages 63–70, New York, NY, USA, 2009. ACM.

[2] Christopher W. Brown and Adam Strzeboński. Black-box/white-box simplification and applications to quantifier elimination. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, ISSAC '10, pages 69–76, New York, NY, USA, 2010. ACM.

[3] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

[4] Clark Barrett. The satisfiability revolution and the rise of smt, 2014.

[5] Alessandro Cimatti. Applicaitons of smt solvers, 2013.

[6] André Platzer, Jan-David Quesel, and Philipp Rümmer. Real world verification. In *Proceedings of the 22Nd International Conference on Automated Deduction*, CADE-22, pages 485–501, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] Miguel Ambrona, Gilles Barthe, and Benedikt Schmidt. Automated unbounded analysis of cryptographic constructions in the generic group model. In *Proceedings of the 35th Annual International Conference on Advances in Cryptology — EUROCRYPT 2016 - Volume 9666*, pages 822–851, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[8] William Denman and César Muñoz. Automated real proving in pvs via metitarski. In *Proceedings of the 19th International Symposium on FM 2014: Formal Methods - Volume 8442*, pages 194–199, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[9] Damian Barsotti, Leonor Prensa Nieto, and Alwen Tiu. Verification of clock synchronization algorithms: Experiments on a combination of deductive tools. *Form. Asp. Comput.*, 19(3):321–341, July 2007.

[10] Ines Lynce Joao Marques-Silva and Sharad Malik. *Conflict-Driven Clause Learning SAT Solvers*, pages 131–153. IOS Press, 1998.

[11] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press.

[12] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, November 2006.

[13] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*, pages 24–84. Springer Vienna, Vienna, 1998.

[14] George E. Collins. *Quantifier elimination for real closed fields by cylindrical algebraic decompostion*, pages 134–183. Springer Berlin Heidelberg, Berlin, Heidelberg, 1975.

[15] Hoon Hong et al. Qepcadb, 2018.

[16] Wolfram mathematica, 2018.

[17] Christopher W. Brown. Tarski, 2018.

[18] Niklas Srensson Niklas En. The minisat page, 2008.